ESCUELA SUPERIOR DE INGENIERÍA

GRADO EN INGENIERÍA INFORMÁTICA

Estrategias de Paralelización Aplicadas a Modelos de Crecimiento Neoplásico

DIRECTOR: Prof. Antonio J. Tomeu Hardasmal
AUTOR: Manuel Francisco Aparicio

15 de junio de 2016

A toda la gente que ha estado siempre a mi lado, día tras día.

Resumen

En biología y medicina, existen dos tipos básicos de experimentación: *in vivo* e *in vitro*. A estas dos se le ha sumado recientemente la simulación por medio de modelos matemáticos y computacionales (experimentos *in silico*). Sin embargo, la mayoría de modelos de simulación tumoral no contemplan soluciones paralelas, por lo que se desaprovecha la mayor parte de la capacidad de cómputo de las máquinas actuales. Nuestro objetivo es analizar la eficiencia de las posibles implementaciones paralelas en dos lenguajes de programación y determinar una estrategia de paralelización adecuada para los modelos de simulación neoplásica.

Abstract

In biology and medicine, there are two kinds of experiments: *in vitro* and *in vivo* experiments. Recently, a new procedure that involves computer simulation has joined this two, known as *in silico* experiments. Remarkably, almost none of the tumor simulators out there have paralelism, leading to a waste of the computational resources of the current computers. Our target is to analize efficiency of the possible parallel implementations in two programming languages and conclude in a suitable parallelization strategy that focus on a tumoral growth model.

Índice general

IX

Índice general IX

Índice de figuras XI

Índice de tablas XIII

1 Introducción 1
- 1.1. Motivación . . . 1
- 1.2. Estado del Arte y Alcance . . . 2
- 1.3. Planificación . . . 2
- 1.4. Por Qué Java . . . 4
- 1.5. Por Qué C++ . . . 4
- 1.6. Conceptos Clave . . . 5

2 Modelo de Simulación Tumoral 7
- 2.1. Investigación y Modelos Neoplásicos . . . 7
- 2.2. La Realidad Biológica: Ciclo Celular Estándar . . . 8
- 2.3. Introducción a los Autómatas Celulares . . . 9
  - 2.3.1. Definición . . . 9
  - 2.3.2. Geometría y Frontera . . . 10
  - 2.3.3. Estados y Transición . . . 10
- 2.4. Modelo de Simulación Tumoral Implementado . . . 11
- 2.5. Entropía . . . 16

3 Desarrollo y Diseño 19
- 3.1. Metodología . . . 19
- 3.2. Objetivos de Diseño . . . 19
- 3.3. Diseño . . . 20
  - 3.3.1. Clase TumorAutomaton . . . 22
  - 3.3.2. Declaración . . . 22
  - 3.3.3. Resumen de Atributos . . . 22

3.3.4. Resumen de Constructores . . . 22
3.3.5. Resumen de Métodos . . . 22
3.3.6. Atributos . . . 23
3.3.7. Constructores . . . 25
3.3.8. Métodos . . . 25
3.4. Casos de Uso . . . 28
3.4.1. Descripción del Caso de Uso: Simular . . . 28
3.4.2. Descripción del Caso de Uso: Reiniciar . . . 29
3.4.3. Descripción del Caso de Uso: Mostrar Información . . . 29
3.4.4. Actores . . . 29
3.5. Modelos de Comportamiento . . . 30
3.5.1. Contrato de Operación: Simular . . . 30
3.5.2. Contrato de Operación: Reiniciar . . . 31
3.5.3. Contrato de Operación: Mostrar Información . . . 31
3.6. Pruebas del Sistema . . . 32

4 Rendimiento **33**
4.1. Consideraciones . . . 33
4.2. Aproximación Secuencial . . . 35
4.3. Problemas Derivados del Enfoque Multihebrado . . . 36
4.4. Cuantificación de la Mejora . . . 37
4.5. Enfoques . . . 37
4.5.1. Cerrojo Único . . . 38
4.5.2. Cerrojo Múltiple . . . 40
4.5.3. Cerrojo en Regiones de Conflicto . . . 44
4.6. Dominio Creciente . . . 46
4.7. Ejecución en Clúster de Apoyo a la Investigación . . . 49

5 Conclusiones y Trabajos Futuros **51**
5.1. Conclusiones . . . 51
5.2. Trabajos Futuros . . . 52

6 Resolución de Conflictos **55**
6.1. Sentido de Iteración . . . 55
6.2. Sincronización en C++ . . . 56

A Análisis de la Entropía **57**
A.1. Tumores Densos . . . 57
A.2. Tumores Dispersos . . . 58
A.3. Parámetros Extremos . . . 59

B Medidas en Máquina Doméstica **61**

C Medidas en Clúster de Supercomputación **79**

D Medidas en Clúster de Supercomputación **89**

E Manual de Usuario **99**
E.1. Introducción . . . 99
E.2. Instalación . . . 99
E.3. Uso del Sistema . . . 99

E.4. Mediciones desde la Línea de Comandos 102
E.4.1. Línea de Comandos: Java 102
E.4.2. Línea de Comandos: C++ 103

Bibliografía **105**

Índice de figuras

1.1. Diagrama de Gantt 3

2.1. Ciclo celular estándar 9
2.2. Vecindad de Moore 10
2.3. Comparativa realidad vs. modelo 12
2.4. Curva de crecimiento Gompertziano 13
2.5. Algoritmo de simulación tumoral 15
2.6. Gráfica de entropía 17

3.1. Diagrama de clases 21
3.2. Diagrama de casos de uso 28
3.3. Diagrama de secuencia del caso de uso Simular 30
3.4. Diagrama de secuencia del caso de uso Reiniciar 31
3.5. Diagrama de secuencia del caso de uso Mostrar Información 31

4.1. Ordenamiento en memoria *column-major*. 34
4.2. Tiempos secuencial 36
4.3. Particiones y no garantización e. m. 37
4.4. Representación de la aproximación con cerrojo único 38
4.5. Speedup cerrojo único 39
4.6. Tiempos cerrojo único 39
4.7. Localidad espacial célula 40
4.8. Representación de la aproximación con múltiples cerrojos 40
4.9. Speedup cerrojo múltiple 43
4.10. Tiempos cerrojo múltiple 43
4.11. Representación de la aproximación con un cerrojo por cada zona fronteriza 44
4.12. Speedup cerrojo frontera 45
4.13. Tiempos cerrojo frontera 46
4.14. Representación de la aproximación con un cerrojo por cada zona fronteriza, con dominio de cómputo creciente 47
4.15. Speedup cerrojos frontera dominio creciente 48
4.16. Tiempos cerrojo frontera dominio creciente 49

6.1. Problemas derivados del sentido de iteración 56

A.1. Gráfica de entropía tumor denso 58

A.2. Gráfica de entropía tumor disperso 58

B.1. Speedup Java cerrojo único estándar 62
B.2. Tiempos Java cerrojo único estándar 63
B.3. Speedup Java cerrojo único alto nivel 63
B.4. Tiempos Java cerrojo único alto nivel 64
B.5. Speedup Java cerrojo múltiple estándar 64
B.6. Tiempos Java cerrojo múltiple estándar 65
B.7. Speedup Java cerrojo múltiple alto nivel 65
B.8. Tiempos Java cerrojo múltiple alto nivel 66
B.9. Speedup Java cerrojo alto nivel frontera 66
B.10. Tiempos Java cerrojo alto nivel frontera 67
B.11. Speedup Java cerrojo estándar frontera 67
B.12. Tiempos Java cerrojo estándar frontera 68
B.13. Speedup Java cerrojo alto nivel frontera dominio creciente 68
B.14. Tiempos Java cerrojo alto nivel frontera dominio creciente 69
B.15. Speedup Java cerrojo estándar frontera dominio creciente 69
B.16. Tiempos Java cerrojo estándar frontera dominio creciente 70
B.17. Speedup de una simulación de 2 años de crecimiento tumoral, en la versión de dominio creciente, implementada en Java. 70
B.18. Tiempos de una simulación de 2 años de crecimiento tumoral, en la versión de dominio creciente, implementada en Java. 71
B.19. Speedup de una simulación de 2 años de crecimiento tumoral, en la versión de dominio creciente, implementada en Java. 71
B.20. Tiempos de una simulación de 2 años de crecimiento tumoral, en la versión de dominio creciente, implementada en Java. 72
B.21. Speedup C++ cerrojo único 72
B.22. Tiempos C++ cerrojo único 73
B.23. Speedup C++ cerrojo múltiple 73
B.24. Tiempos C++ cerrojo múltiple 74
B.25. Speedup C++ cerrojo frontera 74
B.26. Tiempos C++ cerrojo frontera 75
B.27. Speedup C++ cerrojo frontera dominio creciente 75
B.28. Tiempos C++ cerrojo frontera dominio creciente 76
B.29. Speedup de una simulación de 2 años de crecimiento tumoral, en la versión de dominio creciente, implementada en C++. 76
B.30. Tiempos de una simulación de 2 años de crecimiento tumoral, en la versión de dominio creciente, implementada en C++. 77
B.31. Speedup de una simulación de 2 años de crecimiento tumoral, dominio de cómputo creciente. C++ vs Java. 77
B.32. Speedup de una simulación de 2 años de crecimiento tumoral, dominio de cómputo creciente. C++ vs Java. 78

C.1. *Speedup* clúster cerrojo único estándar 80
C.2. Tiempos clúster cerrojo único estándar 80
C.3. *Speedup* clúster cerrojo único alto nivel 81
C.4. Tiempos clúster cerrojo único alto nivel 81
C.5. *Speedup* clúster cerrojo por partición estándar 82
C.6. Tiempos clúster cerrojo por partición estándar 82
C.7. *Speedup* clúster cerrojo por partición alto nivel 83

C.8. Tiempos clúster cerrojo por partición alto nivel 83
C.9. *Speedup* clúster cerrojo para frontera estándar 84
C.10. Tiempos clúster cerrojo para frontera estándar 84
C.11. *Speedup* clúster cerrojo para frontera alto nivel 85
C.12. Tiempos clúster cerrojo para frontera alto nivel 85
C.13. *Speedup* clúster cerrojo por frontera estándar dominio creciente 86
C.14. Tiempos clúster cerrojo por frontera estándar dominio creciente 86
C.15. *Speedup* clúster cerrojo por frontera alto nivel dominio creciente 87
C.16. Tiempos clúster cerrojo por frontera alto nivel dominio creciente 87

E.1. Ventana principal de la aplicación 100
E.2. Ventana principal de la aplicación con una simulación ya iniciada. 101
E.3. Ventana de información adicional. 102

Índice de tablas

4.1. Tiempos según tipo de datos. 35
4.2. Versión de cerrojo único 38
4.3. Versión de cerrojo múltiple 42
4.4. Versión de cerrojo en fronteras 45
4.5. Versión de cerrojos en fronteras con dominio creciente 48

D.1. Tabla clúster cerrojo único estándar 90
D.2. Tabla clúster cerrojo único alto nivel 91
D.3. Tabla clúster cerrojo estándar por partición 92
D.4. Tabla clúster cerrojo alto nivel por partición 93
D.5. Tabla clúster cerrojo estándar zona fronteriza 94
D.6. Tabla clúster cerrojo alto nivel zona fronteriza 95
D.7. Tabla clúster cerrojo estándar zona fronteriza dominio creciente 96
D.8. Tabla clúster cerrojo alto nivel zona fronteriza dominio creciente 97

Capítulo 1

Introducción

1.1. Motivación

Históricamente, la computación siempre ha estado muy ligada a las ciencias. Desde el ENIAC, aplicando métodos de Monte Carlo [1] para el desarrollo de la bomba de hidrógeno, hasta nuestros días, la elevada capacidad de cálculo de un procesador frente a la de un humano ha hecho que sea impensable llevar a término proyectos de investigación sin ellos.

Las investigaciones son caras y no suelen obtenerse resultados a corto plazo. En áreas como la oncología, observar el crecimiento de un tumor o comprobar el efecto de una droga sobre el mismo requiere una alta inversión en tiempo y dinero. Por desgracia, vivimos momentos difíciles: la investigación se está viendo mermada por las constantes reducciones presupuestarias y la disminución de la inversión en ciencia e innovación.

Las simulaciones por ordenador ahorran recursos, tanto temporales como materiales, y permiten experimentar con escenarios complejos con relativa facilidad, aislando a los investigadores de la infinidad de variables presentes en un organismo vivo. La paralelización de los simuladores comienza a ser una condición *sine qua non*, en tanto que los modelos son cada vez más complejos y el rendimiento de los microprocesadores de *core* único ha dejado de aumentar como antes [2].

Durante los dos últimos años, y en el marco de actividades llevadas a cabo como alumno colaborador junto al Prof. Antonio Tomeu, el autor ha trabajado en la paralelización y cuantificación de la mejora de otras tareas, como la convolución de una imagen o el modelado mediante autómata celular de la reacción química de *Belousov-Zhabotinsky*. Siguiendo esta línea de trabajo, se decidió elaborar un modelo de simulación tumoral basado en autómatas celulares, estudiar las diferentes formas de paralelización, cuantificar el rendimiento de cada una y concluir en un método con una buena relación coste/rendimiento, reduciendo al máximo su complejidad, de tal forma que sea fácilmente implementable por un investigador con un mínimo de conocimientos.

1.2. Estado del Arte y Alcance

Existen diferentes técnicas [3] de modelado para simular el crecimiento tumoral; normalmente, se tratan de sistemas matemáticos basados en el uso de ecuaciones diferenciales parciales, sistemas de reacción-difusión, mecánica de medios continuos, etcétera ([4, 5, 6, 7]). Sin embargo, los que mejor reflejan el comportamiento real son de naturaleza estocástica [8], ya que la gran cantidad de variables presentes en un organismo vivo hace que a día de hoy sea imposible determinarlas e introducirlas en un modelo matemático determinista.

En particular, los autómatas celulares demuestran buenos resultados [9, 10, 11] con una relativa facilidad de implementación, y son paralelizables de manera no muy compleja [12]. A pesar de esto, no existen demasiadas referencias [13] en las que se haya paralelizado y medido el incremento del rendimiento de la simulación neoplásica mediante autómatas (aunque sí existen trabajos en otras áreas similares, como la segmentación de imágenes tumorales [14]).

Este trabajo pretende cubrir este aspecto, realizando una comparativa entre diferentes formas de paralelización, a la vez que se desarrolla un modelo básico de simulación tumoral mediante en autómatas celulares.

1.3. Planificación

En la figura 1.1, se presenta el diagrama de Gantt con la planificación temporal del trabajo.

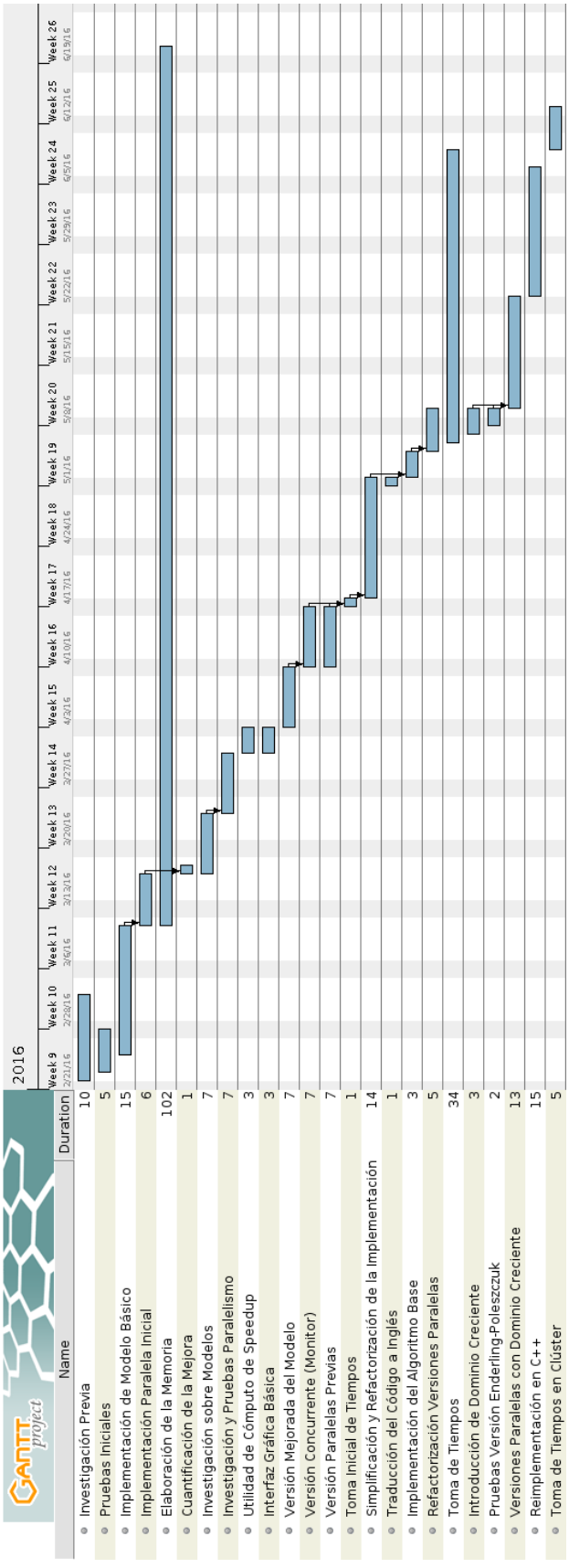

Figura 1.1: Diagrama de Gantt

1.4. Por Qué Java

Java es un lenguaje multiparadigma desarrollado por *Sun Microsystems* en 1995. En su primera versión (JDK 1.0) de 1996 ya incluía soporte a concurrencia [15], incorporando las clases e interfaces básicas (`Thread` y `Runnable`). Podemos decir, por tanto, que se trata de un lenguaje bastante maduro en lo que a paralelismo y concurrencia se refiere.

En la versión 1.5 del kit de desarrollo se realizaron cambios sustanciales en la API, incluyéndose el paquete `java.util.concurrent.*` , y con él utilidades como los ejecutores, esenciales para la correcta manipulación y reutilización de las hebras.

A día de hoy, en la versión 1.8 se incluyen novedades como el *framework fork/join*, una implementación de ejecutor en la que un hilo desocupado puede obtener trabajo de otro (bajo un enfoque *divide and conquer*, divide y vencerás).

Además de las bondades en concurrencia, existen más ventajas que hacen que este lenguaje siga siendo el más popular [16] entre el resto, como ser multiplataforma, gracias a las distintas implementaciones de la máquina virtual, la gran cantidad de usuarios y documentación disponible e, incluso, el rendimiento, gracias a los compiladores JIT existentes. Además, es fácil portar el código a plataformas móviles como Android, puesto que están basadas en este lenguaje.

1.5. Por Qué C++

C++ es, al igual que Java, un lenguaje de programación multiparadigma, creado en 1980 por *Bjarne Stroustrup*. No ha habido soporte oficial a hebras hasta el estándar de 2011 (C++11), por lo que no podemos decir que se trate de un lenguaje maduro en cuanto a concurrencia [17] orientada a objetos se refiere.

Se pueden encontrar funcionalidades básicas, como la clase `std::thread` o distintos contenedores seguros frente a hilos (*thread-safe containers*). Existen los tipos primitivos atómicos (`std::atomic`) al igual que en Java, aunque añaden una mayor flexibilidad al permitir relajar el nivel de atomicidad, de tal forma que las operaciones sean más eficientes.

Hasta la aparición del estándar de 2011, se ha venido usando una implementación de las directivas *POSIX Threads* recogidas en la biblioteca *pthreads* para C/C++. La clase `std::thread` está construida sobre *pthreads*, proporcionando abstracción y facilidades como cerrojos reentrantes, futuros, etcétera.

Se pueden encontrar carencias que, aunque no son funcionales (pueden ser simuladas con las herramientas existentes) incurren en pérdidas de rendimiento; ejemplo de ello es el hecho de tener que crear las hebras en cada ejecución, pues no existen los ejecutores, o la ausencia de barreras, que han de ser implementadas por el programador.

A pesar de todo, C++ es un lenguaje bastante popular [16] y muy eficiente. En sus posteriores versiones [18], tanto el rendimiento como la cantidad de herramientas disponibles en la biblioteca estándar serán mayores. Esperamos obtener buenos resultados ahora y aún mejores en el futuro.

1.6. Conceptos Clave

A continuación, se presentan ciertos conceptos básicos que el lector debe comprender para entender correctamente el documento.

Aislamiento Propiedad que garantiza que un conjunto de instrucciones no afecte a otras que están siendo ejecutadas concurrentemente.

Atomicidad Propiedad que garantiza que un conjunto de instrucciones sea irreducible e indivisible, siendo ejecutadas todas o ninguna, pero nunca de forma parcial.

Barrera Recurso de sincronización que permite bloquear varios hilos hasta que todos hayan llegado a un punto determinado en la ejecución. En el caso de los autómatas celulares, se usan para esperar a la finalización de todas las hebras antes de pasar a computar la siguiente generación.

Cerrojo Mecanismo de sincronización que bloquea el acceso a un recurso compartido o sección crítica, permitiendo así la existencia de exclusión mutua.

Concurrencia Ejecución simultánea (pero no al mismo tiempo) de varias tareas o procesos (falso paralelismo).

Condición de concurso/carrera Existen condiciones de concurso si varios hilos intentan acceder y modificar un recurso concurrentemente.

Consistencia Propiedad de la memoria que garantiza un estado predecible de los datos almacenados (integridad).

Core Núcleo de ejecución de un procesador, que dispone de una unidad aritmético-lógica y una unidad de control (en el caso de arquitecturas MIMD), así como de bancos de registros, cachés, etcétera.

Ejecutor Objeto capaz de ejecutar tareas. Separa el concepto de ejecución independiente de una tarea de las mecánicas de ejecución, creación y planificado de hebras, etcétera.

Entrelazado Secuencia de ejecución de un programa concurrente. Deben existir varias (de lo contrario, el código sería secuencial).

Equidad Se dice que dos hilos son equitativos si todos adquieren acceso al recurso protegido con la misma frecuencia promedia.

Exclusión mutua Condición o técnica que evita que varias hebras accedan a una sección crítica a la vez.

Hebra Unidad de procesamiento planificable por el sistema operativo perteneciente a un proceso que, por tanto, comparte recursos con el resto de hilos creados por el proceso.

Inanición Un hilo presenta inanición si resulta imposible para él acceder al recurso protegido.

Indeterminismo Orden parcial en la ejecución de código concurrente: se conoce qué se realizará, pero no el orden exacto, debido a variables externas no predecibles (por ejemplo, el estado del sistema, las decisiones del planificador, etc.).

Interbloqueo Estado de bloqueo permanente entre varios hilos fruto de un entrelazado específico en la competencia por varios recursos compartidos.

Monitor Clase que encapsula recursos compartidos de forma segura ante la ejecución concurrente, siempre que sean usados mediante los métodos públicos proporcionados.

Paralelismo Especialización de concurrencia. Ejecución simultánea y al mismo tiempo de varias tareas o procesos.

Proceso Ejecución planificable por un sistema operativo de un flujo de instrucciones (programa) en un contexto compuesto por todos los recursos de los que este depende o puede modificar, incluyendo código, variables, memoria, etcétera [19].

Programa Conjunto de sentencias estáticas que realizan una tarea.

Recurso compartido Región de la memoria de un proceso a la que acceden varios hilos concurrentemente, pudiendo provocar inconsistencia si no se protege correctamente.

Reentrancia Propiedad que garantiza que una subrutina pueda llamarse a si misma de forma segura antes de acabar la ejecución inicial. En el caso de los cerrojos, se dice que son reentrantes si permiten a una hebra ejecutar código protegido por un cerrojo previamente capturado por ésta.

Sección crítica Fragmento de código donde se accede a un recurso compartido que puede estar siendo modificado por otro hilo.

Speedup Aceleración. Cuantifica la mejora en la velocidad de ejecución de dos implementaciones de un mismo algoritmo sobre una misma máquina que dispone de unos mismos recursos.

Vivacidad Conjunto de propiedades que hacen que un hilo progrese a pesar de tener que competir por los recursos con el resto.

Capítulo 2

Autómatas Celulares Aplicados a la Simulación Tumoral

2.1. Investigación y Modelos Neoplásicos

En Medicina, existen [20] principalmente dos formas de afrontar una investigación:

- Experimentos clínicos controlados. Se intenta comprobar la validez y el buen funcionamiento de una actuación o medicamento, comparando los efectos y el progreso de dos grupos de personas con patologías idénticas o similares. La clave de estos experimentos es la existencia de un grupo de control al que no se le interviene (al menos de la misma forma), pudiendo así medir si la aplicación del fármaco o procedimiento al otro grupo resulta o no en una mejoría.
- Estudios observacionales. Consisten en una mera observación de grupos de personas por categorías naturales (como la edad, el sexo...) con el fin de encontrar patrones específicos. Al contrario de los ensayos clínicos, estos experimentos no son controlados, por lo que variables externas (como las vitaminas ingeridas por cada persona) pueden influir en los resultados.

Los ensayos clínicos suelen ser más fiables que los estudios observacionales, ya que reducen el sesgo entre el grupo de intervención y el grupo de control.

Cuando realizamos ensayos clínicos, hablamos de experimentos *in vivo*. Un experimento *in vivo* es aquel que se lleva a término dentro de un organismo vivo. Suele ser el más apropiado, puesto que las observaciones e intervenciones realizadas se ajustarán al máximo a la realidad. Sin embargo, en un organismo existe una gran cantidad de variables no controlables que pueden influir en los resultados. De la misma forma, existen escenarios difíciles de estudiar en la realidad, como aquellos que implican un riesgo para la integridad física del sujeto.

Existe otro tipo de experimento que aísla el estudio de los agentes externos. Son los llamados experimentos *in vitro*, realizados en un ambiente controlado fuera de un organismo (por ejemplo, en un tubo de ensayo). Los resultados producidos pueden ser poco exactos, ya que las condiciones en un laboratorio difícilmente pueden emular a las reales. No obstante, permiten crear escenarios impensables en un estudio *in vivo*, por ejemplo, someter el conjunto a temperaturas negativas.

Las ventajas y desventajas que cada uno presenta hacen que sea necesario realizar ambos: si se pretenden obtener resultados, suele realizarse un primer estudio *in vitro* que luego se confirma mediante uno *in vivo*. Si se ha desarrollado una nueva droga para tratar cierta patología, debe pasar por varias fases (como un estudio previo en animales) para posteriormente ser analizada en seres humanos.

Todos estos estudios llevan un coste asociado, tanto monetario como temporal. En los últimos años se han venido usando modelos matemáticos y de simulación por computador como forma de obtener datos previos a una investigación *in vitro* que ayuden al desarrollo y pronóstico de la intervención [21]. Los modelos matemáticos permiten especificar condiciones difícilmente alcanzables de otra forma, a la vez que proporcionan escenarios totalmente controlados.

No todos los modelos matemáticos son igual de válidos. Los modelos basados en poblaciones, como el *Gompertziano*, no describen el comportamiento a nivel de células particulares, sino que estiman parámetros relativos a la población, limitando demasiado las conclusiones que un investigador puede obtener. Otras aproximaciones, como aquellas basadas en ecuaciones diferenciales parciales o sistemas de reacción-difusión no captan la realidad estocástica presente en el crecimiento de una neoplasia [8].

Es, por todo ello, por lo que quizá sean los autómatas celulares no deterministas la mejor forma de ajustar la realidad a una simulación computerizada. Cada celda del autómata representa una célula tumoral, y su estado se rige mediante reglas y distribuciones de probabilidad, por lo que se puede ajustar cada escenario modificando la definición de la regla y las extracciones aleatorias de la distribución probabilística.

A estos experimentos basados en simulación tumoral computerizada se les ha venido denominando experimentos *in silico*, en clara referencia a los chips de silicio usados actualmente.

2.2. La Realidad Biológica: Ciclo Celular Estándar

El ciclo celular es un conjunto ordenado de sucesos que conducen al crecimiento y posterior división de una célula. Se divide en dos fases principales, la interfase (estado de no división) y la mitosis (estado de división). A su vez, la interfase se divide en tres estados principales, G_1, S y G_2.

- G_1: Primera fase del ciclo celular, en la que la célula crece y es sensibles a inhibidores del crecimiento, como algunas hormonas. Tiene una duración de entre 6 y 12 horas, y en ese tiempo se duplica su volumen y masa.
- S: Segunda fase del ciclo celular. Se produce la replicación del ADN. Tiene una duración de 10 a 12 horas.
- G_2: Tercera fase del ciclo celular. Se continúa la síntesis de proteínas y ARN. Tiene una duración de entre 3 y 4 horas.
- M: Cuarta fase del ciclo celular (mitosis). Se produce la división celular. Tiene una duración de aproximadamente 30 minutos.

No todas las células se replican continuamente. En ocasiones, una célula puede entrar en estado G_0. Este estado es considerado independiente del ciclo celular, y se

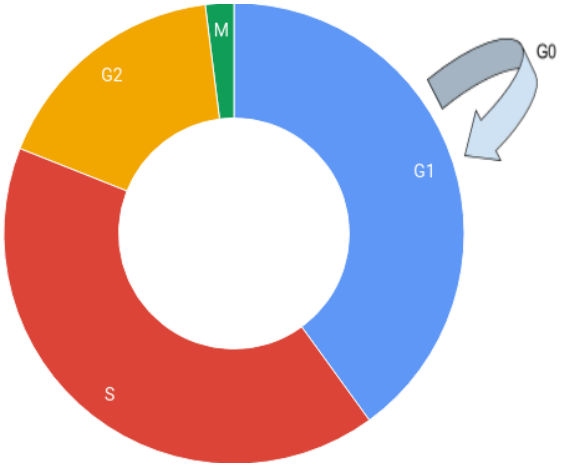

Figura 2.1: Ciclo celular estándar

puede producir después de una mitosis, pudiendo finalmente desembocar en el estado G_1. Es un estado en el que la célula permanece quieta (*quiescente* o *senesciente*).

2.3. Introducción a los Autómatas Celulares

Los autómatas celulares fueron definidos inicialmente por *John von Neumann* y *Stanislaw Ulam* en la década de 1950. Se pueden encontrar diferentes definiciones en la bibliografía [22], pero todas tienen la misma base.

2.3.1. Definición

Usaremos la definición general que se puede encontrar en [23]. Un autómata celular (CA) queda definido por una cuadrupla $(Q, \varepsilon, N^I, \delta)$ donde:

- Q es una rejilla discreta de celdas (o células) con una determinada condición de frontera.
- ε es un conjunto de estados finito, normalmente pequeño, que las células pueden adoptar.
- N^I es un conjunto finito de células que definen la vecindad con la que interactuará cada celda.
- δ es una regla (función de transición) que define la dinámica de estados de cada celda.

2.3.2. Geometría y Frontera

Un espacio celular $Q \subset \mathbb{R}^d$ es un conjunto de células que cubre homogéneamente un espacio euclídeo d-dimensional. Una célula es un polígono regular. Cada célula queda identificada por su posición $r \in Q$. Para cualquier coordenada $r \in Q$, la retícula que define la vecindad más cercana $N_b(r)$ es una lista de células vecinas definida por

$$N_b(r) = \{r + c_i : c_i \in N_b, i = 1, ..., b\} \subseteq Q$$

donde b es el número de células que conforman la vecindad.

En nuestro caso, usaremos la conocida *Vecindad de Moore*, que queda definida como

$$N_8^I = \{(-1,-1), (-1,0), (-1,1), (0,-1), (0,1), (1,-1), (1,0), (1,1)\}$$

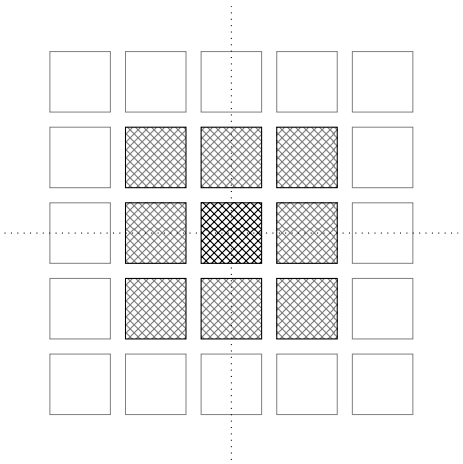

Figura 2.2: Vecindad de Moore

2.3.3. Estados y Transición

A cada célula $r \in Q$ se le asocia un estado $s(r) \in \varepsilon$, siendo s una aplicación $s : Q \to \varepsilon$.

Una configuración global $\sigma \in \varepsilon^{|Q|}$, donde $|Q|$ denota el número total de células, es determinada por el estado de todas las células de la rejilla Q.

Una configuración local es un vector σ_M determinado por el estado de un conjunto ordenado de células denotado por $M \subset Q$, tal que

$$\forall r_i \in M, \quad \sigma_M = (s(r_1), s(r_2), ..., s(r_i)), \quad M \subset Q$$

Se dice que una configuración τ es sucesora de una configuración σ, y se denota por $\sigma \vdash \tau$, si

$$\forall r \in Q, \quad \delta(\sigma(r)) = \tau(r)$$

siendo δ una aplicación

$$\delta : \varepsilon^\mu \to \varepsilon, \quad \text{donde } \mu = |N_b^I|$$

La regla es espacialmente homogénea, por lo que el estado resultante no depende de la posición r que ocupe la célula.

Nuestro autómata celular no es determinista, por lo que la función de transición δ no llevará a un único estado posible para los parámetros de entrada, sino que responderá a una distribución de probabilidad W, de forma que la definición de nuestra configuración sucesora se verá alterada como sigue

$$\sigma \vdash \tau \longleftrightarrow \forall r \in Q, \quad \delta(\sigma(r)) = \tau(r) \quad \text{con probabilidad} \quad W(\sigma_{N(r)} \to \tau_{N(r)})$$

donde

$$z^j \in \varepsilon : \tau(r) = \begin{cases} z^1 & \text{con probabilidad} \quad W(\sigma_{N(r)} \to z^1) \\ z^2 & \text{con probabilidad} \quad W(\sigma_{N(r)} \to z^2) \\ \vdots & \vdots \quad\quad\quad\quad \vdots \\ z^{|\varepsilon|} & \text{con probabilidad} \quad W(\sigma_{N(r)} \to z^{|\varepsilon|}) \end{cases}$$

siendo $W(\sigma_{N(r)} \to z^j)$ una distribución probabilística independiente del tiempo que especifica la probabilidad de alcanzar el estado z^j dada la configuración de la vecindad $\sigma_{N(r)}$.

2.4. Modelo de Simulación Tumoral Implementado

La mayoría de autómatas celulares que modelan el crecimiento neoplásico siguen el mismo esquema de base: cada célula del autómata es un individuo de la población de células tumorales, pudiendo tener dos estados básicos, viva (1) o muerta (0). Una célula viva puede migrar, proliferar mediante mitosis, morir espontáneamente o permanecer latente conforme a diferentes probabilidades previamente especificadas.

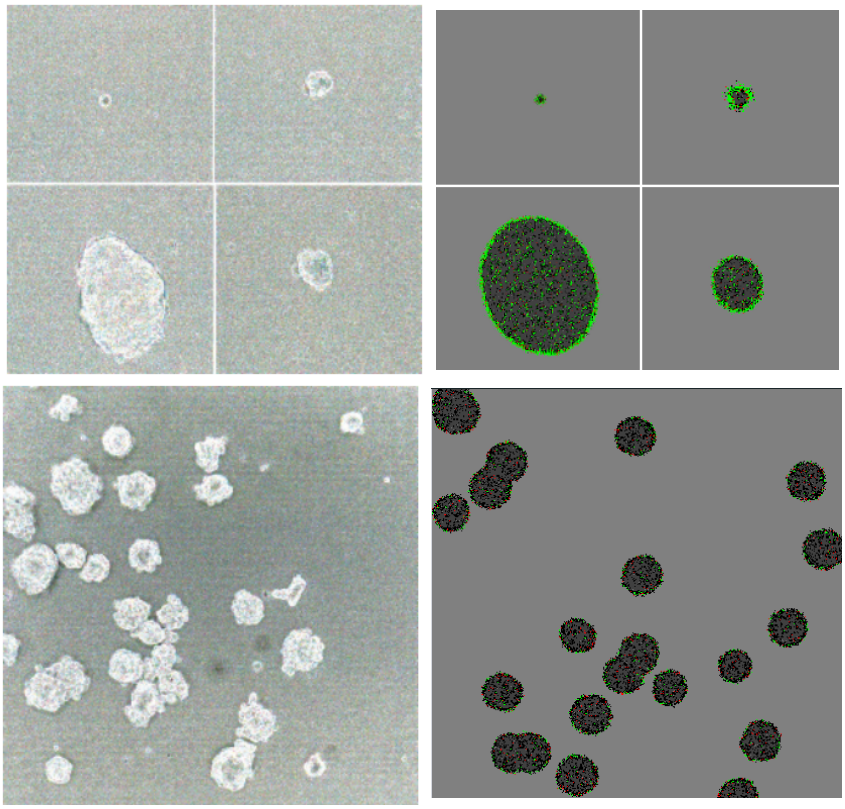

Figura 2.3: Comparativa entre dos crecimientos reales (izquierda) *in-vitro* (tomados de [24]) frente a crecimientos simulados por nuestro modelo (derecha). Podemos ver un alto grado de similitud.

Las células tumorales pueden ser *stem* (madres), con capacidad de proliferación infinita, o *no-stem*, que mueren al dividirse un número determinado de veces.

Nuestro modelo resulta de tomar las mejores características de los modelos presentados en [3] y [25]. Consideramos tres estados básicos: muerta (0), latente (1) o viva (2).

- Una célula muerta no puede realizar ninguna acción, y es un hueco libre al que las células aledañas podrán proliferar o migrar.

- Una célula latente podrá morir espontáneamente, pero no podrá proliferar ni migrar a huecos en la vecindad.

- Una célula viva podrá morir espontáneamente, migrar o proliferar a una celda marcada como vacía en su vecindad.

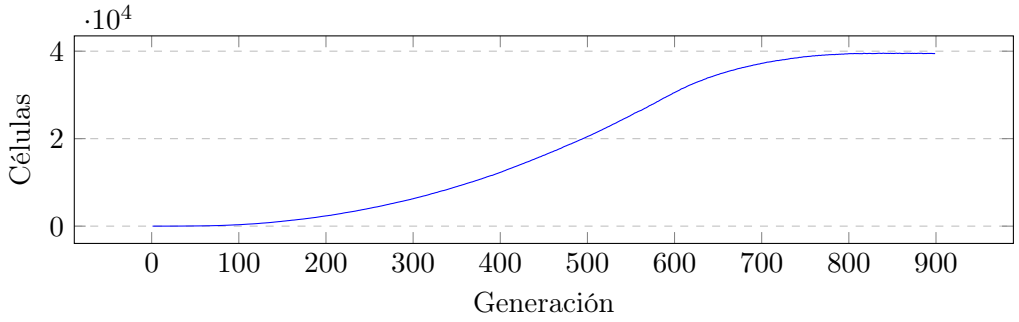

Figura 2.4: Crecimiento *Gompertziano* [26] de la población del tumor, para un dominio tisular de 200^2 células. Esta curva es característica de los tumores sólidos, como el de mama o próstata.

La apoptosis o muerte espontánea se rige por una distribución de probabilidad. Una célula, latente o viva, morirá si se realiza una extracción de la distribución de probabilidad W_S que la regula y está fuera del segmento $[P_S, 1]$, donde P_S es la probabilidad de supervivencia de una célula.

$$\delta_S(\sigma_{N(r)}) = \begin{cases} 0 & \text{con probabilidad} \quad W_S(\sigma_{N(r)} \to 0) \\ s(r) & \text{con probabilidad} \quad W_S(\sigma_{N(r)} \to 1) \end{cases}$$

La proliferación o mitosis celular queda regida mediante una distribución probabilística, la existencia de espacio en la vecindad y de un número de señales de proliferación PH suficientemente grande (mayor o igual a NP). Una célula viva podrá proliferar a una posición libre u en la vecindad si existe tal posición, el número de señales de proliferación PH de esa célula es mayor o igual al numero mínimo NP de señales para proliferar y se realiza una extracción de la distribución de probabilidad W_P que la regula, quedando ésta dentro del segmento $[0, P_P)$, siendo P_P la probabilidad de que la célula prolifere.

$$\delta_P(\sigma_{N(t)}) = \begin{cases} 0 & \text{con probabilidad} \quad W_P(\sigma_{N(u)} \to 0) \\ 2 & \text{con probabilidad} \quad W_P(\sigma_{N(u)} \to 2) \end{cases}$$

La posición u estará ahora ocupada, o no, por una célula resultado de la mitosis celular de la célula r. La célula r pasará a estar muerta si se ha excedido el número máximo $\rho_{\text{máx}}$ de veces que una célula puede proliferar.

La migración celular queda regulada mediante una distribución de probabilidad y la existencia de espacio libre alrededor. Una célula viva r podrá migrar a una posición libre en la vecindad u si existe tal posición y se realiza una extracción de la distribución de probabilidad W_M que la regula y queda dentro del segmento $[0, P_M)$, siendo P_M la probabilidad de que una célula migre.

$$\delta_M(\sigma_{N(t)}) = \begin{cases} 0 & \text{con probabilidad} \quad W_M(\sigma_{N(u)} \to 0) \\ 2 & \text{con probabilidad} \quad W_M(\sigma_{N(u)} \to 2) \end{cases}$$

La posición u será ahora ocupada por una célula resultado de la migración de la célula que estaba en r, quedando ahora la posición r desocupada.

Si por azar, una célula viva no muere, ni migra, ni prolifera, ha de permanecer inactiva para la generación dada. Si una célula no muere, pero no puede proliferar ni migrar por no existir espacio libre en la vecindad, su estado pasa a ser latente (1), con lo que en lo sucesivo únicamente podrá morir espontáneamente, hasta que sea despertada por una célula aledaña que deje hueco.

A partir de lo anterior, hemos adaptado los algoritmos propuestos en [3] y [25], obteniendo el que representamos en el siguiente pseudocódigo y mediante un diagrama de flujo en la figura 2.5 de la página 15.

```
Establecer condiciones iniciales
desde i=1 hasta k
    para_cada celula r
        si r != MUERTA
            rrs = aleatorio();

            si rrs < Ps
                si hayEspacioVecindad(r)
                    rrp = aleatorio()
                    r.ph = r.ph + 1

                    si (rrp < Pp ^ r.ph >= NP)
                        posicion = calcularDireccionProliferacion(r)
                        proliferar(posicion)
                    en_otro_caso
                        rrm = aleatorio()

                        si rrm < Pm
                            posicion = calcularDireccionMigracion(r)
                            migrar(r, posicion)
                        fin_si
                    fin_si
                en_otro_caso
                    latente()
                fin_si
            en_otro_caso
                apoptosis()
                despertarVecindad()
            fin_si
        fin_si
    fin_para
fin_desde
```

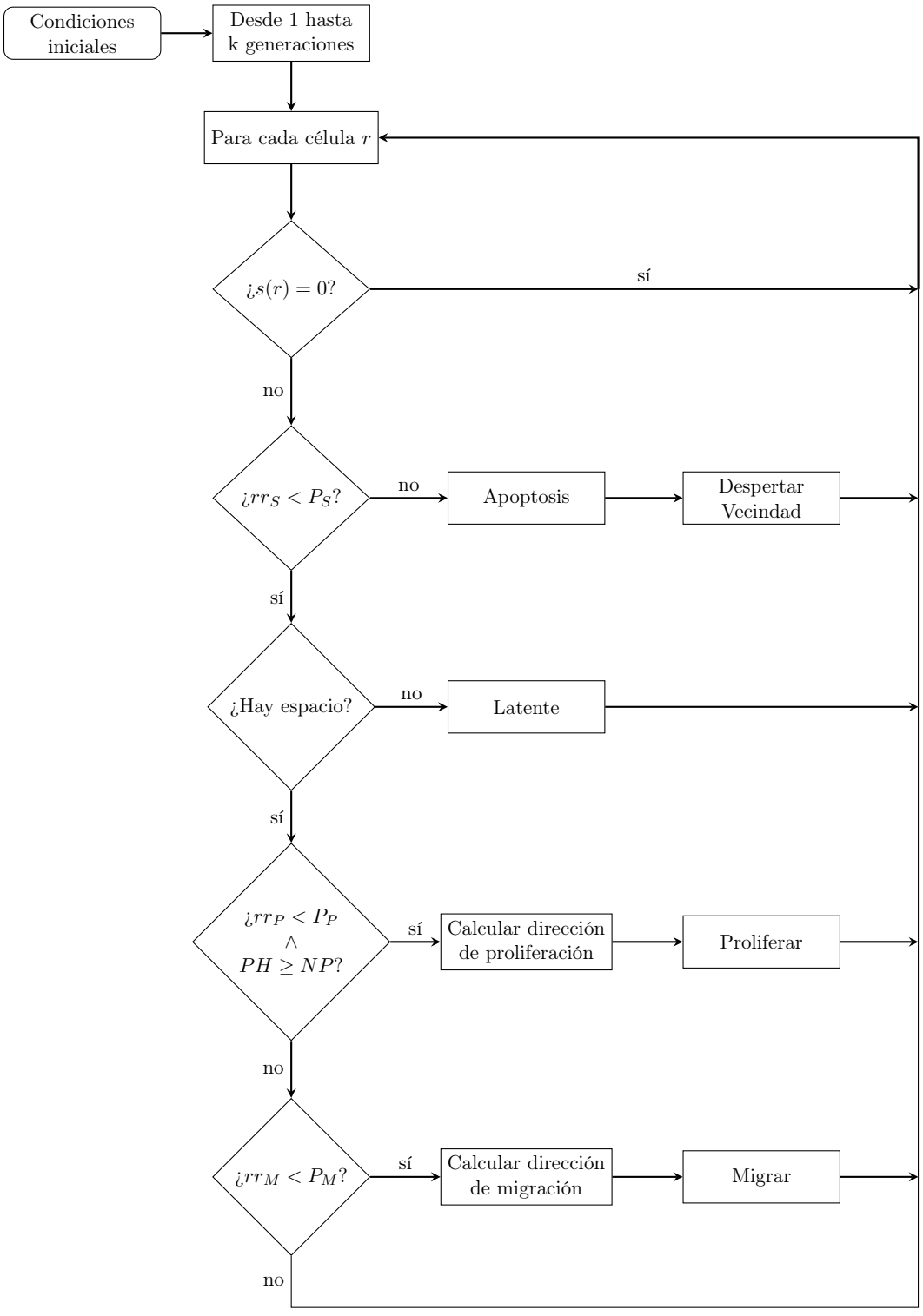

Figura 2.5: Algoritmo de simulación tumoral

2.5. Entropía

Tal y como se recoge en [27], podemos definir la entropía como

$$H(x) = -\sum_{i=1}^{n} p_i \log p_i$$

donde $p_1, p_2, ..., p_n$ son las probabilidades de suceso de los n posibles estados del sistema.

La entropía nos permite medir el grado de incertidumbre de un modelo estocástico. De entre sus propiedades, hemos seleccionado las siguientes:

- H es continua en p_i.
- Si todas las probabilidades p_i son iguales, H será una función monótona creciente de n (cuanto mayor es el grado de semejanza de las probabilidades, más incierto será el resultado).
- $H = 0$ si y solo si todos los p_i excepto uno son cero. Esto es, únicamente es posible una salida del sistema estocástico, por lo que no hay incertidumbre. En caso contrario, H es positiva.
- La incertidumbre de una unión de eventos es menor o igual que la suma de las incertidumbres individuales.
- Todo intento de ecualización de las probabilidades incurren en un crecimiento de la entropía.

En nuestro modelo se presentan tres estados posibles ($\{0, 1, 2\}$). La probabilidad p_0 viene dada por

$$p_0 = \frac{\text{n. de células muertas}}{\text{n. total de células}}$$

mientras que la probabilidad p_1 será

$$p_1 = \frac{\text{n. de células latentes}}{\text{n. total de células}}$$

y la probabilidad p_2

$$p_1 = \frac{\text{n. de células vivas}}{\text{n. total de células}}$$

para cada instante de tiempo discreto t en la ejecución de nuestro modelo.

A partir de lo anterior y dadas la ecuación y características descritas de la entropía, así como de los conocimientos de comportamiento de los tumores sólidos, podemos esperar que la curva de entropía se divida en tres fases:

- Fase monótona creciente. El tumor irá creciendo por lo que la probabilidad de que una célula esté muerta irá disminuyendo en favor de células vivas y latentes. La entropía llegará a su máximo cuando el número de células muertas iguale al de vivas y al de latentes.

- Fase monótona decreciente. El dominio tisular comenzará a saturarse; será más probable encontrar una célula latente o viva que una célula muerta, por lo que el grado de incertidumbre bajará.

- Fase de monotonía constante, con algún oscilamiento debido a la apoptosis. El dominio tisular se ha saturado y el tumor está en su máximo tamaño. Las células muertas que podremos encontrar serán debido a apoptosis, y el número de células vivas y latentes estará aproximadamente igualado. Se producen fluctuaciones con una clara tendencia hacia una constante. En el caso de no existir la apoptosis (cuando $P_S = 1$ y todas las células son *stem*), se alcanzará un punto donde todas las células estén latentes, *ergo* la entropía será nula.

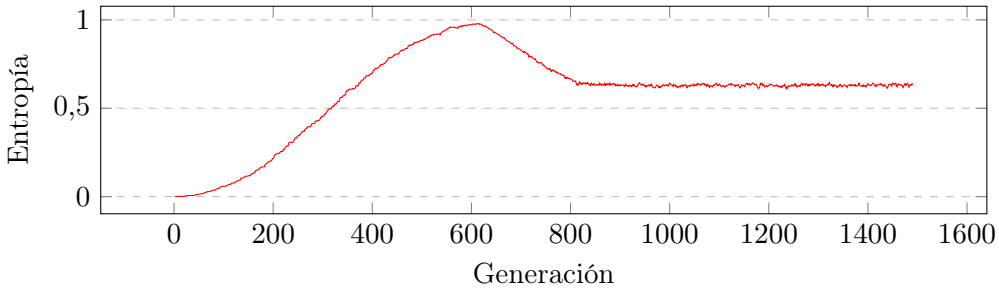

Figura 2.6: Gráfica de entropía para un tumor sólido de 200^2 células. Pueden observarse las tres fases descritas y las fluctuaciones de la curva.

Se ha realizado un análisis algo más detallado sobre el comportamiento de la entropía en el apéndice A.

Capítulo 3

Desarrollo y Diseño

3.1. Metodología

Se ha seguido un enfoque de desarrollo iterativo, concretamente, se ha usado una adaptación propia de la metodología ágil conocida como *programación extrema* [28]. Hemos considerado este enfoque como el más adecuado dado la naturaleza poco definida de nuestro trabajo.

La *programación extrema* caracteriza programar como la actividad clave en un proyecto de software. Está pensada para equipos de trabajo pequeños en el contexto de software cuyos requisitos pueden cambiar en cualquier instante. Hay quien la caracteriza como *la metodología del sentido común*. Su máxima es la simplicidad: se realizará la implementación mínima que cumpla su cometido. Esto es perfectamente conjugable con el desarrollo paralelo, puesto que un excesivo nivel de abstracción puede incurrir en peores aceleraciones y tiempos de ejecución. Contempla pruebas unitarias y de integración en el software.

Ha de hacerse hincapié en la adaptabilidad del proyecto más que en la previsibilidad y por tanto planificación, ya que no se conocen los requisitos del sistema ni se sabe cómo van a evolucionar. Es por ello que se necesita una metodología que contemple cambios sobre la marcha.

Todos los roles han sido asumidos por el autor conforme a las orientaciones del director.

3.2. Objetivos de Diseño

Se pretenden alcanzar los siguientes objetivos:

- Desarrollo mediante un autómata celular de un modelo de simulación de neoplasias paralelo altamente personalizable, que optimice el uso de recursos computacionales de la máquina.

- Implementación de una interfaz gráfica de usuario que muestre el desarrollo de la simulación tumoral e información adicional, como las gráficas de crecimiento y entropía.

3.3. Diseño

Es requisito indispensable simplificar la arquitectura del modelo de simulación de neoplasias, por lo que se ha realizado en una única clase. Para la capa de presentación, se han usado tres clases, siendo una de ellas la que encapsula el modelo de simulación.

En el diagrama de la figura 3.1 puede observarse el resultado de la última iteración de nuestro programa.

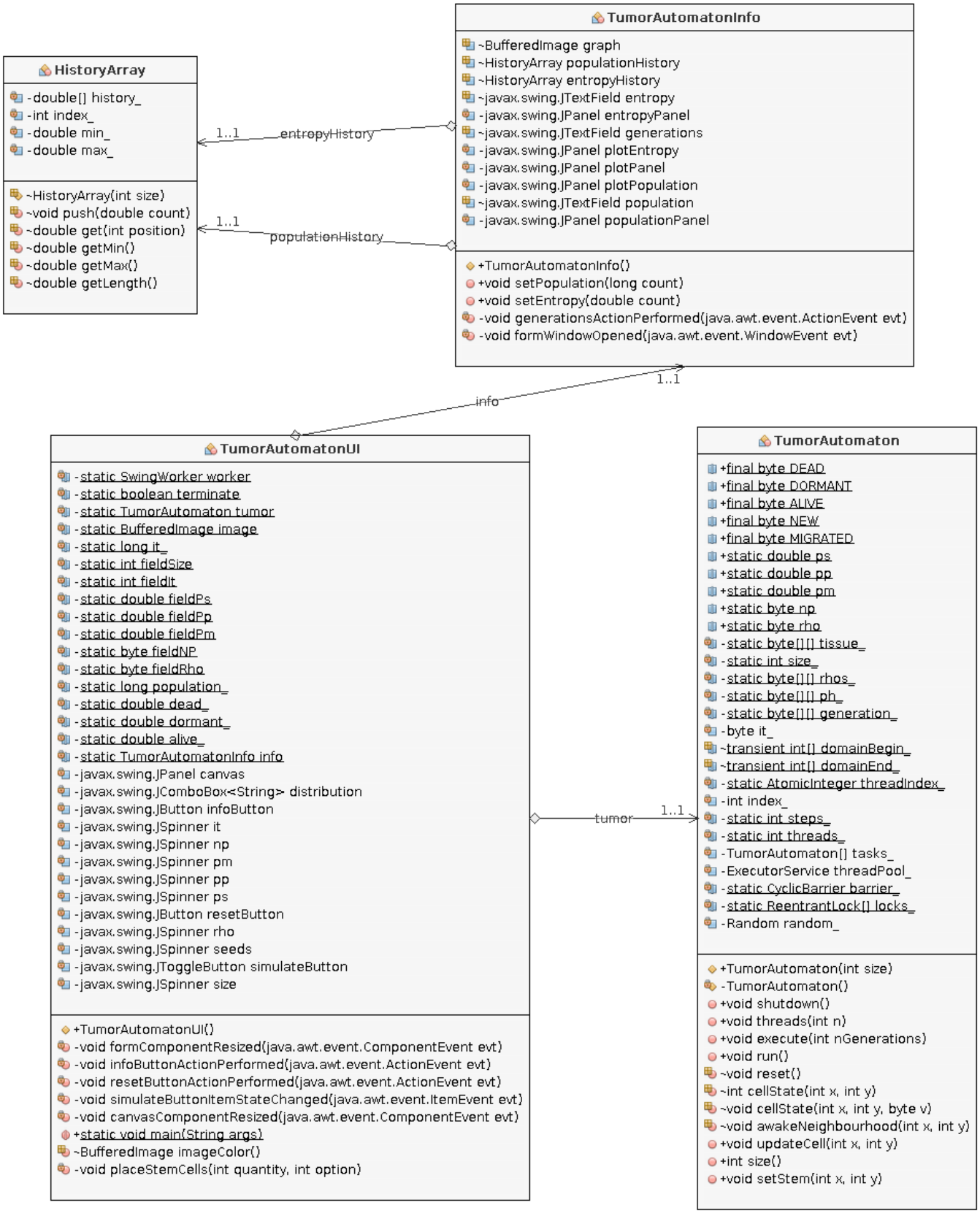

Figura 3.1: Diagrama de clases

3.3.1. Clase TumorAutomaton

Se presenta a continuación la clase de simulación tumoral. Se incluye una descripción pormenorizada de los atributos y métodos.

3.3.2. Declaración

```
1 public class TumorAutomaton
2  extends java.lang.Object implements java.lang.Runnable
```

3.3.3. Resumen de Atributos

ALIVE Estado de una célula viva.
barrier_ Barrera de sincronización.
DEAD Estado de una célula muerta.
domainBegin_ Inicio del dominio de ejecución.
domainEnd_ Fin del dominio de ejecución.
DORMANT Estado de una célula latente.
generation_ Control de generaciones.
index_ Índice del hilo.
it_ Etiqueta de la generación actual.
locks_ Colección de cerrojos.
MIGRATED Estado de una célula que acaba de migrar.
NEW Estado de una célula recién creada.
np Número de señales necesarias para proliferar.
ph_ Número de señales de proliferación de cada célula.
pm Probabilidad de migración.
pp Probabilidad de proliferación.
ps Probabilidad de supervivencia.
random_ Generador de números aleatorios.
rho Número de veces que una célula puede proliferar sin morir.
rhos_ Número de veces que puede proliferar cada célula.
size_ Tamaño del dominio tisular.
steps_ Pasos de tiempo discreto a ejecutar.
tasks_ Colección de tareas.
threadIndex_ Contador de índices de hilos.
threadPool_ Ejecutor.
threads_ Número de tareas.
tissue_ Dominio tisular.

3.3.4. Resumen de Constructores

TumorAutomaton() Constructor de tareas.
TumorAutomaton(int) Crea un nuevo autómata celular con el tamaño especificado.

3.3.5. Resumen de Métodos

awakeNeighbourhood(int, int) Despierta a las células latentes de la vecindad de (x, y)

cellState(int, int) Devuelve el estado de la célula (x, y).
cellState(int, int, int) Modifica el estado de la célula (x, y).
execute(int) Ejecuta el autómata durante un número determinado de generaciones.
reset() Reinicia el autómata celular.
run() Método ejecutable.
setStem(int, int) Coloca una célula stem en las coordenadas (x, y).
shutdown() Intenta parar el ejecutor y espera hasta que este haya terminado.
size() Devuelve el tamaño del dominio tisular.
threads(int) Configura el autómata para ser ejecutado en paralelo.
updateCell(int, int) Ejecuta la regla del autómata para la célula (x, y).

3.3.6. Atributos

- `public static final byte DEAD`
  - Estado de una célula muerta.
- `public static final byte DORMANT`
  - Estado de una célula latente.
- `public static final byte ALIVE`
  - Estado de una célula viva.
- `public static final byte NEW`
  - Estado de una célula recién creada.
- `public static final byte MIGRATED`
  - Estado de una célula que acaba de migrar.
- `public static double ps`
  - Probabilidad de supervivencia.
- `public static double pp`
  - Probabilidad de proliferación.
- `public static double pm`
  - Probabilidad de migración.
- `public static byte np`
  - Número de señales necesarias para proliferar.
- `public static byte rho`
  - Número de veces que una célula puede proliferar sin morir.
- `private static byte[][] tissue_`
  - Dominio tisular.

- `private static int size_`
  - Tamaño del dominio tisular.
- `private static byte[][] rhos_`
  - Número de veces que puede proliferar cada célula.
- `private static byte[][] ph_`
  - Número de señales de proliferación de cada célula.
- `private static byte[][] generation_`
  - Control de generaciones.
- `private byte it_`
  - Etiqueta de la generación actual.
- `static volatile int[] domainBegin_`
  - Inicio del dominio de ejecución.
- `static volatile int[] domainEnd_`
  - Fin del dominio de ejecución.
- `private static java.util.concurrent.atomic.AtomicInteger threadIndex_`
  - Contador de índices de hilos.
- `private int index_`
  - Índice del hilo.
- `private static int steps_`
  - Pasos de tiempo discreto a ejecutar.
- `private static int threads_`
  - Número de tareas.
- `private TumorAutomaton[] tasks_`
  - Colección de tareas.
- `private java.util.concurrent.ExecutorService threadPool_`
  - Ejecutor.
- `private static java.util.concurrent.CyclicBarrier barrier_`
  - Barrera de sincronización.
- `private static java.util.concurrent.locks.ReentrantLock[] locks_`
  - Colección de cerrojos.
- `private java.util.Random random_`
  - Generador de números aleatorios.

3.3.7. Constructores

- **TumorAutomaton**

```
1 private TumorAutomaton()
```

  - **Descripción**
    Constructor de tareas.

- **TumorAutomaton**

```
1 public TumorAutomaton(int size)
```

  - **Descripción**
    Crea un nuevo autómata celular con el tamaño especificado.
  - **Parámetros**
    - `size` – Tamaño del dominio tisular.

3.3.8. Métodos

- **awakeNeighbourhood**

```
1 void awakeNeighbourhood(int x,int y)
```

  - **Descripción**
    Despierta a las células latentes de la vecindad de (x, y)
  - **Parámetros**
    - `x` – Coordenada en el eje x.
    - `y` – Coordenada en el eje y.

- **cellState**

```
1 int cellState(int x,int y)
```

  - **Descripción**
    Devuelve el estado de la célula (x, y).
  - **Parámetros**
    - `x` – Coordenada en el eje x.
    - `y` – Coordenada en el eje y.
  - **Devuelve** – Estado de la célula (x, y).

- **cellState**

```
1 void cellState(int x,int y,byte v)
```

  - **Descripción**
    Modifica el estado de la célula (x, y).
  - **Parámetros**
    - `x` – Coordenada en el eje x.
    - `y` – Coordenada en el eje y.
    - `v` – Nuevo estado de la célula (x, y).

- **execute**

```
1 public void execute(int nGenerations)
```

  - **Descripción**
    Ejecuta el autómata durante un número determinado de generaciones.
  - **Parámetros**
    - `nGenerations` – Número de generaciones a computar

- **reset**

```
1 void reset()
```

  - **Descripción**
    Reinicia el autómata celular.

- **run**

```
1 public void run()
```

  - **Descripción**
    Método ejecutable. Calcula el número de generaciones especificado en el atributo 'steps_', dentro de la partición del hilo que lo ejecute.

- **setStem**

```
1 public void setStem(int x,int y)
```

  - **Descripción**
    Coloca una célula *stem* en las coordenadas (x, y).
  - **Parámetros**
    - `x` – Coordenada en el eje x.
    - `y` – Coordenada en el eje y.

- **shutdown**

```
1 public void shutdown()
```

  - **Descripción**
    Intenta parar el ejecutor y espera hasta que este haya terminado.

- **size**

```
1 public int size()
```

  - **Descripción**
    Observador del tamaño del dominio tisular.
  - **Devuelve** – Tamaño del dominio tisular.

- **threads**

```
1 public void threads(int n)
```

  - **Descripción**
    Configura el autómata para ser ejecutado en paralelo. Si se especifica 0, el autómata se ejecutará secuencialmente.
  - **Parámetros**
    - `n` – Número de tareas a ejecutar

- **updateCell**

```
1 public void updateCell(int x,int y)
```

  - **Descripción**
    Ejecuta la regla del autómata para la célula (x, y).
  - **Parámetros**
    - `x` – Coordenada en el eje x.
    - `y` – Coordenada en el eje y.

3.4. Casos de Uso

Se presentan a continuación los casos de uso, que quedan recogidos como puede verse en el diagrama de la figura 3.2

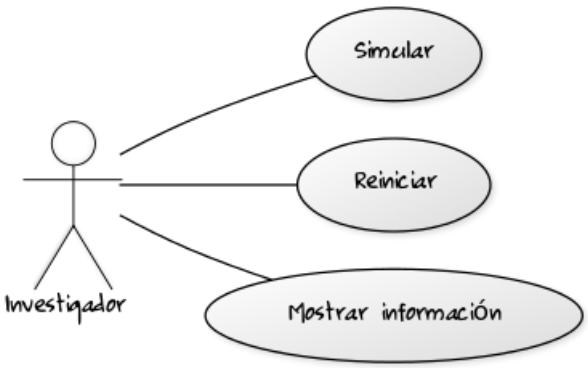

Figura 3.2: Diagrama de casos de uso

3.4.1. Descripción del Caso de Uso: Simular

- Caso de uso: Simular.
- Descripción: Simula el crecimiento de un tumor con los parámetros introducidos.
- Actores: Investigador, Sistema.
- Precondiciones: El programa se encuentra iniciado. Se han especificado parámetros válidos.
- Postcondiciones: Se simula el crecimiento de un tumor.
- Escenario principal

  1. El usuario introduce los parámetros que quiere simular.
  2. El usuario pulsa el botón de simular.
  3. El sistema instancia un objeto de la clase `TumorAutomaton` con dichos parámetros.
  4. El sistema introduce las semillas *stem*.
  5. El sistema simula el crecimiento del tumor, hasta que el usuario vuelva a pulsar el botón de nuevo.

- Escenarios alternativos

  3a. Ya existe un objeto de la clase tumor instanciado.

  1. El sistema ajusta los parámetros que hayan sido modificados.
  2. El sistema continúa con el paso 5.

3.4.2. Descripción del Caso de Uso: Reiniciar

- Caso de uso: Reiniciar.
- Descripción: Reinicia el tumor a su estado inicial.
- Actores: Investigador, Sistema.
- Precondiciones: El programa se encuentra iniciado. Se han especificado parámetros válidos.
- Postcondiciones: Se reinician los atributos del objeto tumor.
- Escenario principal
    1. El usuario pulsa sobre el botón de reiniciar.
    2. El sistema reinicia el objeto tumor.
    3. El sistema reconfigura los parámetros.
    4. El sistema introduce las semillas *stem*.
- Escenarios alternativos

    2a. El objeto tumor no está instanciado.
        1. El sistema instancia la clase `TumorAutomaton`.
        2. El sistema continúa con el paso 3.

3.4.3. Descripción del Caso de Uso: Mostrar Información

- Caso de uso: Mostrar Información.
- Descripción: Muestra la ventana de información adicional.
- Actores: Investigador, Sistema.
- Precondiciones: El programa se encuentra iniciado.
- Postcondiciones: Se abre la ventana de información adicional.
- Escenario principal
    1. El usuario pulsa sobre el botón de información.
    2. El sistema abre la ventana de información.
- Escenarios alternativos

    2a. La ventana ya está abierta
        1. El sistema no hace nada.

3.4.4. Actores

Se contempla un único rol (Investigador) para todos los usuarios, con acceso a todas las funciones.

3.5. Modelos de Comportamiento

A partir de los casos de uso anteriores se establece el modelo de comportamiento del sistema. Se establecen diferentes contratos de operación junto a diagramas de secuencia que explican el comportamiento.

3.5.1. Contrato de Operación: Simular

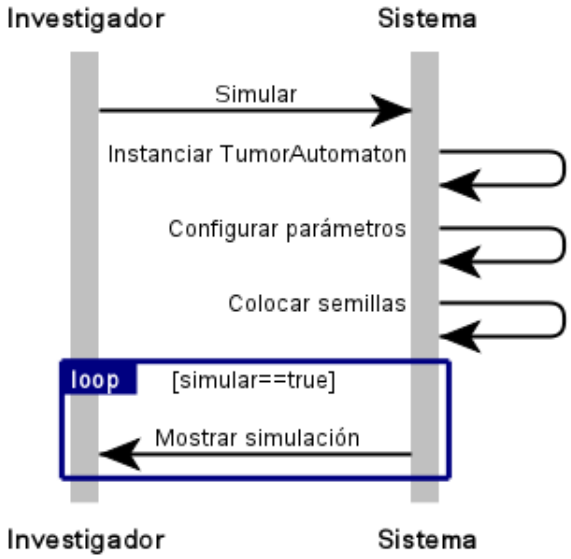

Figura 3.3: Diagrama de secuencia del caso de uso Simular

- **Operación**: Simular

- **Responsabilidades**: Instanciar la clase `TumorAutomaton`, configurar los parámetros, colocar semillas e iniciar la simulación.

- **Precondiciones**: Los parámetros introducidos son válidos.

- **Postcondiciones**: Se actualiza la interfaz gráfica mostrando el estado del tumor en cada iteración.

3.5.2. Contrato de Operación: Reiniciar

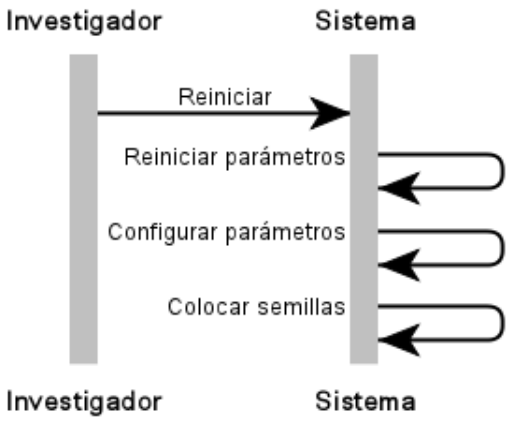

Figura 3.4: Diagrama de secuencia del caso de uso Reiniciar

- **Operación**: Reiniciar
- **Responsabilidades**: Reiniciar parámetros, configurarlos y colocar las semillas.
- **Precondiciones**: Ninguna.
- **Postcondiciones**: Se actualiza la interfaz gráfica mostrando el estado inicial del tumor.

3.5.3. Contrato de Operación: Mostrar Información

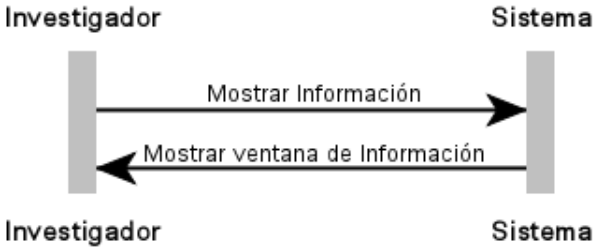

Figura 3.5: Diagrama de secuencia del caso de uso Mostrar Información

- **Operación**: Mostrar información
- **Responsabilidades**: Mostrar ventana de información.
- **Precondiciones**: Ninguna.
- **Postcondiciones**: Se abre la ventana de información.

3.6. Pruebas del Sistema

Durante los ciclos del desarrollo se han realizado pruebas unitarias y de integración no automatizadas. Se ha comprobado el buen funcionamiento de los métodos y la integración de los mismos con la interfaz gráfica. Para comprobar el paralelismo y correcto particionado de la matriz se han usado células *debug* (valor -1), que no influían en el comportamiento del algoritmo.

De la misma forma, se ha comparado el comportamiento del programa frente a escenarios conocidos justificados por otros autores [3, 26] (por ejemplo, la curva de crecimiento *Gompertziano* que caracteriza a los tumores sólidos).

Capítulo 4

Rendimiento

4.1. Consideraciones

A la hora de implementar un autómata celular hay que tener ciertos detalles en cuenta si queremos maximizar el rendimiento. Para realizar optimizaciones hasta los niveles más bajos hay que conocer el funcionamiento del lenguaje, el modelo de memoria que usa, etc.

Cosas tan básicas y simples como realizar operaciones de preincremento (`++i`) en vez de postincremento (`i++`) cuando no sea necesario mantener una copia del valor original de la variable puede incurrir en una mejora sustancial de tiempo en código que es iterado bastantes veces.

En esta sección queremos hacer hincapié en el modelo de memoria usado por C++ y Java, y en su forma de almacenar matrices, ya que acceder correctamente a los datos nos permitirá elevar la tasa de aciertos en la caché del procesador, disminuyendo así las esperas a la memoria RAM.

Las arquitecturas actuales contemplan varios niveles en la jerarquía de memoria. En el nivel más cercano al procesador podemos encontrar los registros. Por encima de ellos se encuentra la caché, que puede tener distintos niveles (L1, L2, L3...), la memoria principal (RAM), y la memoria secundaria (almacenamiento masivo). Por lo general, cuanto más alta esté la memoria en la jerarquía, mayor es la latencia y menor la velocidad.

Los datos siempre se recuperan siguiendo la jerarquía en memoria: se recurre a los niveles más bajos; si no se encuentra, se solicita al nivel superior. Cada vez que se solicita un dato a la memoria caché, y el dato se encuentra presente, se produce un acierto. Una elevada tasa de aciertos ahorra multitud de ciclos de reloj en espera a que el siguiente nivel en la jerarquía sirva los datos.

La mayoría de cuellos de botella que se presentan en los computadores actuales tienen que ver con el acceso a memoria. En la actualidad, el diseño del hardware se centra en la optimización de las cachés, los accesos a ella, la previsión y carga adelantada de datos (*prefetching*), el correcto segmentado del *pipeline*...

Es, por todo ello, totalmente necesario que nuestro código se adapte lo mejor posible al funcionamiento de las cachés: cuando se solicita un dato en memoria, se entregan automáticamente ése y los bytes más cercanos (*burst mode*), porque es bastante probable que sean requeridos con posterioridad. A partir de esto y sabiendo

cómo organizan C++ y Java los datos en memoria (figura 4.1), es lógico pensar que el orden correcto para iterar el autómata es primero las columnas y luego las filas.

```
1 for (int i = 0; i < N; ++i)
2     for (int j = 0; j < N; ++j)
3         automata[i][j] ...
```

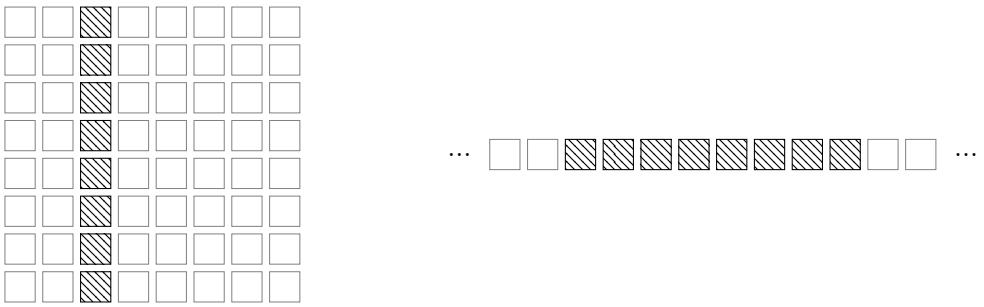

Figura 4.1: Ordenamiento en memoria *column-major*.

Ejemplo

Tenemos la matriz

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

que será almacenada en memoria por columnas, es decir,

...	1	3	2	4	...

Supongamos que cada línea de caché puede almacenar dos enteros, y que cada vez que se solicita un dato a memoria, ésta envía también el siguiente. Si recorriésemos la matriz por filas, tendríamos:

$$m_{0,0} \text{ (fallo en caché)} \rightarrow m_{1,0} \text{ (fallo en caché)} \rightarrow$$
$$\rightarrow m_{0,1} \text{ (fallo en caché)} \rightarrow m1,1 \text{ (fallo en caché)}$$

Es fácil ver que recorrer por columnas es más beneficioso, en tanto que

$$m_{0,0} \text{ (fallo en caché)} \rightarrow m_{0,1} \text{ (acierto)} \rightarrow$$
$$\rightarrow m_{1,0} \text{ (fallo en caché)} \rightarrow m1,1 \text{ (acierto)}$$

De la misma forma, es necesario evitar, en la medida de lo posible, estructuras y contenedores dinámicos enlazados. Los nodos no suelen ocupar posiciones adyacentes de memoria, por lo que los fallos de la caché aumentan. Colecciones como `ArrayList` en Java y `std::vector` en C++ permiten un mejor funcionamiento de la caché que sus equivalentes enlazados (`List` y `std::list` respectivamente).

4.2. Aproximación Secuencial

La implementación secuencial del modelo no difiere en demasía del diagrama de flujo presentado en la página 15. Se dispone de cuatro matrices:

- Matriz tejido, que representa el autómata celular en sí. Cada celda contiene el estado de la célula que ocupa dicha posición.
- Matriz ρ, contiene el parámetro ρ de cada célula. Este parámetro regula el número de veces que una célula puede proliferar hasta su muerte.
- Matriz PH, que contiene el número de señales de proliferación que una célula ha recibido.
- Matriz *generación*, contiene información sobre la generación en la que ha sido creada una célula. Necesaria para evitar deformidades hacia la dirección de iteración (ver 6.1).

Es de esperar que el tiempo de ejecución dependa del tipo de datos usado en estas matrices. No es lo mismo usar un entero de 64 bits que un único byte para representar los estados posibles del autómata. El dominio es tan pequeño que no presenta problema alguno para ser definido por 8 bits.

Es posible que, inicialmente, un programador elija el tipo `int` espontáneamente, inherente a sus costumbres. Sin embargo, la diferencia en la elección de un buen tipo de datos no es despreciable.

Dimensión	`long`	`int`	`byte`
2000	18,62	17,29	17,17
4000	69,06	68,40	68,07
6000	154,57	152,17	144,78
8000	308,45	282,85	258,22

Tabla 4.1: Tiempos en segundos para diferentes tipos de datos. Implementación secuencial del modelo de simulación, en un procesador *dual core* Intel® Core™ i5-4300U @2,90GHz

Puede observarse una mejora de hasta un 16 %. Esta puede considerarse como la primera optimización. Se trata de una medida básica muy fácil de llevar a cabo, por lo que no habría motivo aparente para elegir un tipo de tamaño superior a un byte.

En la figura 4.2 pueden comprobarse los tiempos de ejecución para la versión secuencial. De ahora en adelante, las pruebas presentadas se realizan en Java, versión 1.8.0_77 de la OpenJDK, y en C++, compilador *g++* versión 5.3.0, estándar C++11, nivel 3 de optimización, sobre ArchLinux (64 bits) bajo el kernel 4.4.5-1 de Linux. La máquina dispone de un procesador AMD Athlon™ II X4 640 @3GHz, *quad-core*, 2 MB de caché L2, 512 KB de caché L1, sin caché L3, junto a 8GB de memoria RAM DDR2-1200 *dual-channel*. El lector puede ver y analizar el resto de pruebas en los apéndices.

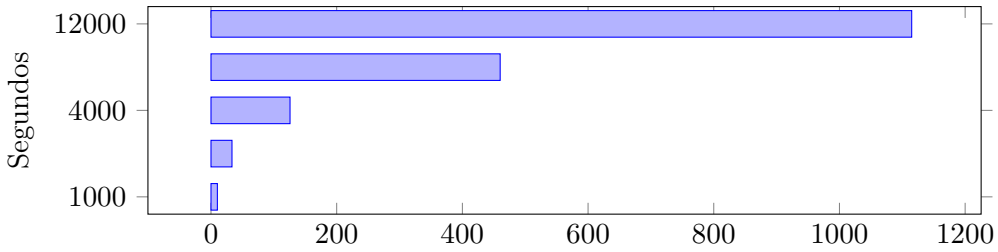

Figura 4.2: Tiempos de la solución secuencial con una carga paramétrica del modelo de $P_S = 1$, $P_P = 0{,}8$, $P_M = 0{,}2$, $NP = 5$ y $\rho = 2$, sobre un tejido cuadrado de diferentes tamaños, durante 1000 pasos de tiempo discreto. Se realizan a partir de una única semilla *stem* en el centro del tejido. La máquina dispone de un procesador *quad-core* @3GHz y 8GB de RAM. Puede ver el resto de características en el apéndice B.

4.3. Problemas Derivados del Enfoque Multihebrado

Introducir varios hilos de ejecución en un programa sin consideración alguna puede acarrear problemas.

Es necesario distribuir bien el trabajo entre las hebras, y para ello, es fundamental saber cuántas tareas lanzar: dado que existe un coste asociado a la creación y ejecución de un hilo, hay un número óptimo suputable mediante la *ecuación de Subramanian* [29]

$$N_t = \frac{N_{nd}}{1 - C_b}$$

donde N_t es el número de tareas a lanzar, N_{nd} es el número de núcleos de los que dispone nuestra unidad de cómputo y C_b es un coeficiente de bloqueo. El coeficiente de bloqueo representa el tanto por uno en tiempo que un procesador permanece a la espera durante la ejecución total de un programa, y es determinable mediante el análisis matemático del mismo para un procesador dado. Debido a la alta complejidad que esto supone, no son pocos los casos en los que se determina empíricamente.

Por regla general, los programas que realizan cómputos intensivos en la CPU tienen un coeficiente muy cercano a 0. En cambio, si disponemos de programas con latencias altas, como aquellos que realizan tareas de red, el coeficiente será muy cercano a 1. De esta forma, es fácil deducir que

$$\lim_{C_b \to 0} \frac{N_{nd}}{1 - C_b} = N_{nd} \qquad \lim_{C_b \to 1} \frac{N_{nd}}{1 - C_b} = \infty$$

y por ello, dado que nuestro modelo requiere un trabajo principalmente computacional, podemos asumir que el número óptimo de tareas será la cantidad de núcleos de los que dispongamos, *ergo* el tejido tumoral ha de ser divido equitativamente entre ellos (paralelismo de datos de grano grueso). Dividir la matriz es tan simple como repartir el número de filas o columnas que procesa cada tarea. Como nuestros lenguajes de programación utilizan un ordenamiento por columnas, lo más sensato sería repartirlas, de tal forma que cada hebra pueda iterar su partición con localidad espacial, para favorecer la tasa de aciertos de la caché.

Sin embargo, tener N_{nd} hilos modificando el tejido simultáneamente puede generar inconsistencia en los recursos compartidos. Por ende, resulta indispensable

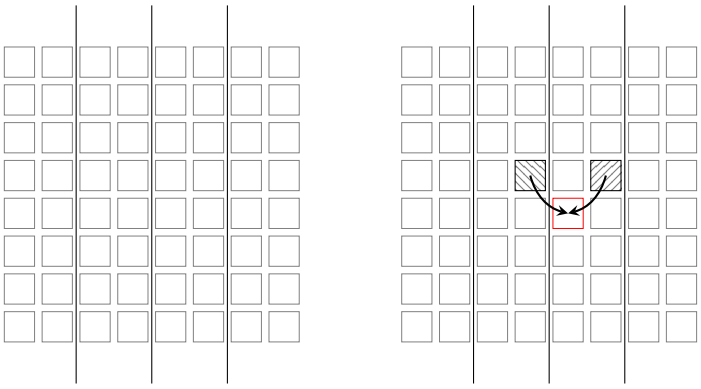

Figura 4.3: Ejemplo de particiones de datos de una matriz (izquierda) y esquema de las consecuencias de la no garantización de la exclusión mutua (derecha).

protegerlos. En la siguiente sección ofrecemos las distintas formas de hacerlo, junto a una comparativa de recursos temporales e incremento del rendimiento.

4.4. Cuantificación de la Mejora

Para cuantificar el factor de mejora al paralelizar el algoritmo, hemos usado el índice *speedup* (aceleración):

$$S = \frac{T(\phi)}{T(m)}$$

donde S es el *speedup*, $T(\phi)$ es el tiempo del mejor algoritmo secuencial y $T(m)$ el tiempo de nuestro algoritmo paralelo.

- El índice *speedup* está en niveles normales si se encuentra en el intervalo $[1, N_{nd}]$, siendo N_{nd} el número de núcleos disponibles.
- Si se encuentra entre $(-\infty, 1)$ nos encontramos ante una implementación peor que la secuencial.
- Si se encuentra entre $(N_{nd}, +\infty)$ hablamos de un *speedup* hiperlineal. No suele ser habitual, pero puede encontrarse en determinados problemas [30, 31] en arquitecturas específicas de memoria caché, o en trabajos con altas latencias (por ejemplo, aquellos en los que hay que esperar una respuesta de una petición en la red. El tiempo de la ejecución secuencial implica sumar todas las latencias; en una ejecución paralela, pueden realizarse todas las peticiones a la vez. El tiempo de espera vendría dado por la latencia máxima).

4.5. Enfoques

Es sencillo proteger de forma básica los recursos compartidos. Con el mero hecho de bloquear al resto de hilos mientras se esté realizando una escritura conseguimos asegurar la consistencia de la memoria. No obstante, existen diferentes formas de tomar los bloqueos que detallamos a continuación.

4.5.1. Cerrojo Único

La aproximación de cerrojo único implica ejecutar en exclusión mutua todas aquellas secciones críticas que existan en el código, es decir, todas aquellas secciones que realicen una escritura o que su resultado dependa del estado de las células aledañas.

Cada vez que se aplique el algoritmo de simulación tumoral (figura 2.5) a una célula, habrá que ejecutar en exclusión mutua a partir de la bifurcación *¿Hay espacio?*. De la misma forma, el proceso de *Apoptosis* también deberá ser ejecutado con el resto de hilos bloqueados.

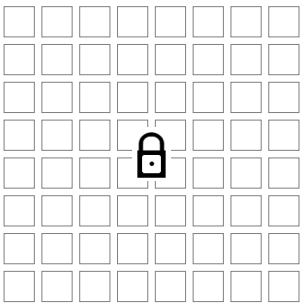

Figura 4.4: Representación de la aproximación con cerrojo único

En las figuras 4.5 y 4.6 pueden observarse algunos de los datos recogidos. De la misma forma, en la tabla 4.2 pueden verse cifras para un dominio tisular de 8000^2 células durante 1000 generaciones.

Número de tareas	Tiempo Java (seg.)	Tiempo C++ (seg.)
1	460.15	153.91
2	258.72	153.90
4	131.05	81.30
6	117.10	85.14
8	114.76	82.45
10	116.66	83.63
12	113.84	81.71
14	115.53	82.43
16	113.64	81.84

Tabla 4.2: Ejecución de la solución de cerrojo único con una carga paramétrica del modelo de $P_S = 1$, $P_P = 0{,}8$, $P_M = 0{,}2$, $NP = 5$ y $\rho = 2$, sobre un tejido cuadrado de 8000 celdas de lado, durante 1000 pasos de tiempo discreto. Se realizan a partir de una única semilla *stem* en el centro del tejido. La máquina dispone de un procesador *quad-core* @3GHz y 8GB de RAM. Puede ver el resto de características en el apéndice B.

Es evidente que C++ es más rápido que Java. Hay que tomar la curva de *speedup* con cautela, pues Java denota mejor *speedup*. Debemos recordar que este índice es relativo a la versión secuencial. En la ejecución de hebra única, C++ necesita menos de la mitad del tiempo que precisa Java. Este último mejora drásticamente con la inclusión de hilos, no pudiéndose decir lo mismo de C++, a pesar de que los resultados siguen siendo mejores.

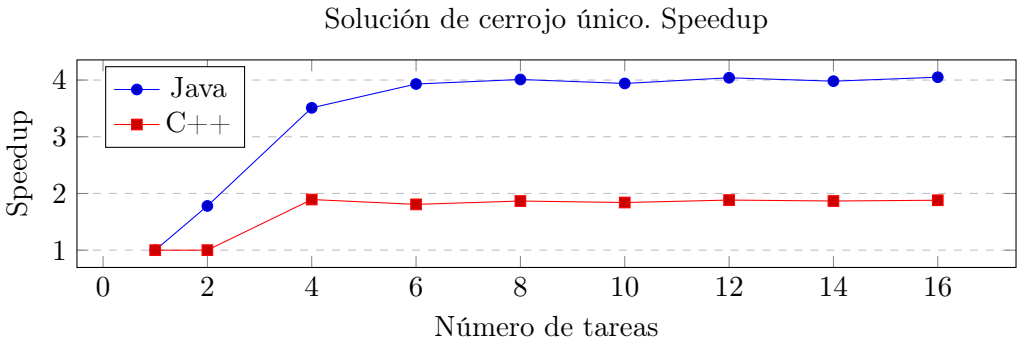

Figura 4.5: Speedup de la solución de cerrojo único con una carga paramétrica del modelo de $P_S = 1$, $P_P = 0{,}8$, $P_M = 0{,}2$, $NP = 5$ y $\rho = 2$, sobre un tejido cuadrado de 8000 celdas de lado, durante 1000 pasos de tiempo discreto. Se realizan a partir de una única semilla *stem* en el centro del tejido. La máquina dispone de un procesador *quad-core* @3GHz y 8GB de RAM. Puede ver el resto de características en el apéndice B.

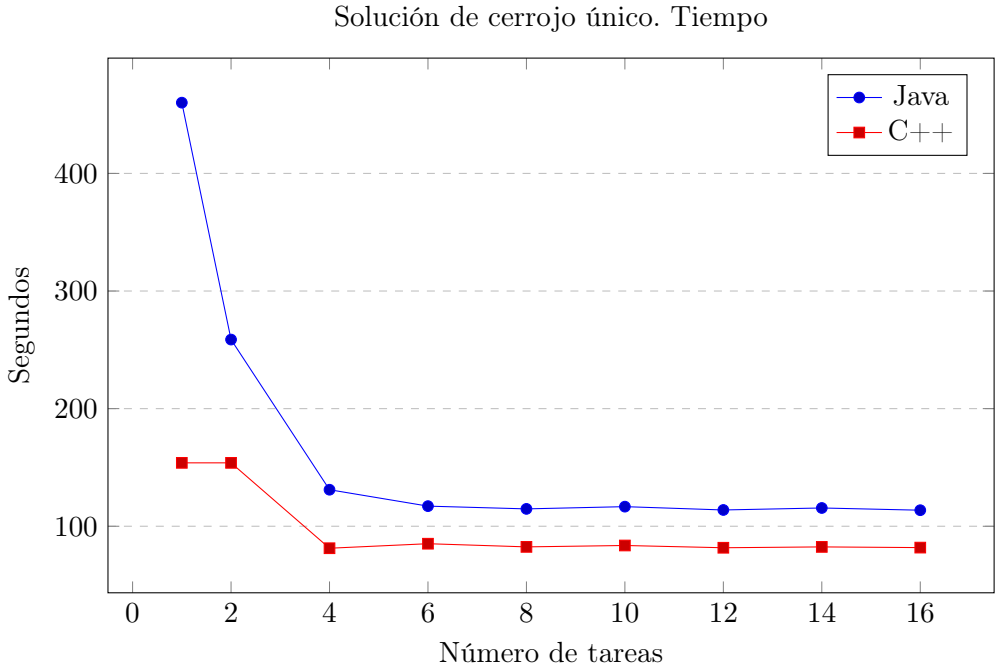

Figura 4.6: Tiempos de la solución de cerrojo único con una carga paramétrica del modelo de $P_S = 1$, $P_P = 0{,}8$, $P_M = 0{,}2$, $NP = 5$ y $\rho = 2$, sobre un tejido cuadrado de 8000^2 células, durante 1000 pasos de tiempo discreto. Se realizan a partir de una única semilla *stem* en el centro del tejido. La máquina dispone de un procesador *quad-core* @3GHz y 8GB de RAM. Puede ver el resto de características en el apéndice B.

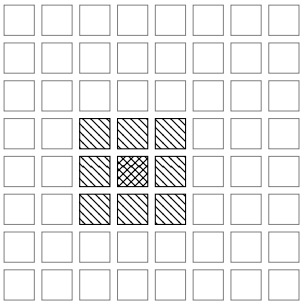

Figura 4.7: Localidad espacial de una célula. Solo podrá verse modificada por las células resaltadas

Los tiempos paralelos son muy similares entre ambos lenguajes, para asombro del autor, aunque esto ya había sido constatado por otros autores [32, 33, 34]. A pesar de la capa de la máquina virtual, los resultados obtenidos demuestran la alta optimización de los recursos de concurrencia de Java.

Sin embargo, no resulta complicado darse cuenta de que los bloqueos son excesivos. Nuestro modelo es un algoritmo con una localidad bien definida. Una célula únicamente se verá afectada por las celdas inmediatamente a su alrededor, en su vecindad. Por consiguiente, no tiene sentido bloquear un hilo que se encuentra realizando cambios fuera del segmento de datos de otro.

4.5.2. Cerrojo Múltiple

Para evitar bloquear un hilo que está trabajando fuera del segmento de datos que intentamos procesar, se propone un enfoque de cerrojo múltiple. En él, cada segmento de datos dispondrá de su propio cerrojo, con lo que un hilo únicamente se bloqueará si intenta escribir en el segmento de datos de otro hilo que se encuentra procesándolo.

Cada hebra podrá procesar su segmento de datos sin depender de lo que el resto de tareas estén realizando en sus respectivas particiones del dominio tisular. Cada vez que un hilo entre en su sección crítica, capturará el cerrojo de la submatriz que esté procesando; si en algún momento invade una partición aledaña, capturará también el cerrojo de ésta, viéndose obligado a esperar si otro hilo está en proceso.

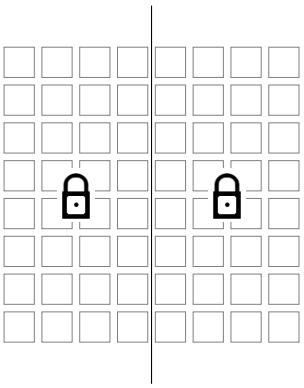

Figura 4.8: Representación de la aproximación con múltiples cerrojos

Esto plantea un problema difícil de detectar si no se tiene claro el comportamiento no determinado y los diferentes entrelazados que puede sufrir la ejecución de nuestro programa. Veámoslo con un ejemplo.

Ejemplo

- Un hilo A procesa una célula cercana al segmento de datos del hilo B.
- Para ello, captura el cerrojo L_A, realiza las operaciones necesarias, y concluye que es preciso escribir en el segmento de datos del hilo B.
- Mientras tanto, el hilo B se encuentra procesando una célula muy cercana al segmento de datos del hilo A, por lo que tiene su cerrojo L_B bloqueado.
- El hilo A intentará capturar el cerrojo L_B, sin éxito, por lo que se bloqueará hasta que éste se libere.
- El hilo B, determina preciso escribir en el segmento de datos de A. Intentará capturar el cerrojo L_A, sin éxito, porque A lo mantiene bloqueado.

El comportamiento descrito es conocido como interbloqueo (*deadlock*). Cuando un interbloqueo se presenta en un programa de relativa complejidad, resulta muy difícil detectarlo: no se lanza ninguna excepción, ni se percibe ningún error. Para la máquina, la ejecución es correcta, pero el programa no acabará nunca.

Para lidiar con este problema, hemos estimado necesario modificar levemente el algoritmo. Si se decide realizar una migración o proliferación a una partición distinta a la que corresponde el cerrojo que mantenemos capturado, guardamos los cambios **hasta después de haber liberado el que actualmente tenemos**. Esto implica posponer la escritura hasta un momento posterior en el que ya se haya determinado la acción a realizar, se haya liberado el cerrojo, y se esté en condiciones de capturar otro sin que se produzca el indeseado *deadlock*.

Ejemplo

- Un hilo A procesa una célula cercana al segmento de datos del hilo B.
- Para ello, captura el cerrojo L_A, realiza las operaciones necesarias, y concluye que es preciso escribir en el segmento de datos del hilo B.
- Mientras tanto, el hilo B se encuentra procesando una célula muy cercana al segmento de datos del hilo A, por lo que tiene su cerrojo L_B bloqueado.
- El hilo A guardará la acción a realizar en variables auxiliares, saldrá de su sección crítica (liberando su cerrojo L_A) e intentará capturar el cerrojo L_B, sin éxito, puesto que B lo mantiene bloqueado.

- El hilo B, determina preciso escribir en el segmento de datos de A. Pospondrá la escritura mediante el uso de variables auxiliares, saldrá de su sección crítica e intentará capturar el cerrojo L_A. Como A ya liberó el cerrojo, B no encontrará problema para escribir en la partición de A.
- Mientras tanto y como consecuencia de la liberación de L_B, A ya se habrá despertado, habrá capturado L_B y se encontrará realizando la escritura.
- Ambos hilos acaban la ejecución correctamente.

Con este problema solucionado, la implementación del algoritmo es trivial y no presenta apenas diferencias con la versión anterior. Presentamos los resultados obtenidos en la tabla 4.3 y en las figuras 4.9 y 4.10

Número de tareas	Tiempo Java (seg.)	Tiempo C++ (seg.)
1	517.81	182.47
2	258.93	137.89
4	131.03	71.27
6	137.58	76.12
8	114.69	74.44
10	116.46	75.41
12	113.89	73.47
14	115.61	74.69
16	113.50	73.79

Tabla 4.3: Ejecución de la solución de cerrojo múltiple con una carga paramétrica del modelo de $P_S = 1$, $P_P = 0{,}8$, $P_M = 0{,}2$, $NP = 5$ y $\rho = 2$, sobre un tejido cuadrado de 8000 celdas de lado, durante 1000 pasos de tiempo discreto. Se realizan a partir de una única semilla *stem* en el centro del tejido. La máquina dispone de un procesador *quad-core* @3GHz y 8GB de RAM. Puede ver el resto de características en el apéndice B.

Observando las figuras de esta sección, podemos ver tiempos similares y *speedups* casi idénticos a los ofrecidos en el punto 4.5.1. C++ sigue siendo superior, y se prueba más sensible a la mejora que Java.

De nuevo, para el asombro del autor, no hemos obtenidos resultados significativos de la mejora descrita. Llegados a este punto, estimamos que esto era debido a que seguíamos tomando el mismo número de bloqueos, aunque sobre diferentes objetos, a lo que hay que sumar la introducción de lógica adicional para seleccionar el cerrojo adecuado en cada momento.

A partir de lo anterior y volviendo a tener en cuenta la localidad de la aplicación del algoritmo, restringida a la *vecindad de Moore* de una célula (ver figura 2.2), es posible inferir que se toman bloqueos en demasía. No tiene sentido bloquear un segmento de datos si el hilo que lo procesa se encuentra realizando cambios en una posición en la que es imposible que otro hilo cercano acceda, porque queda fuera del

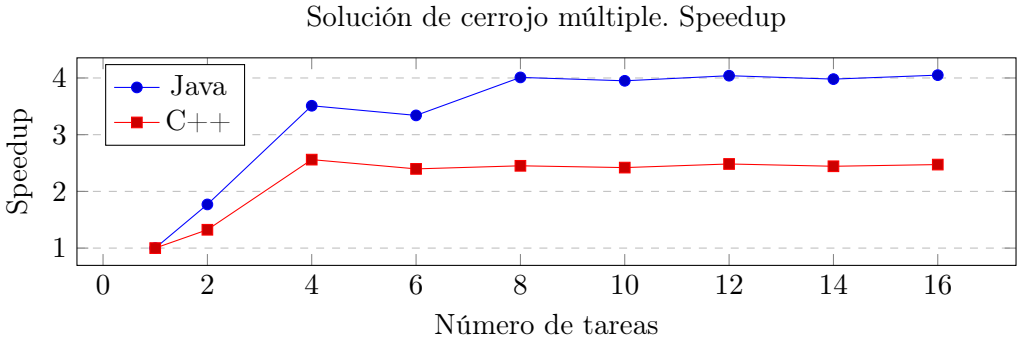

Figura 4.9: Speedup de la solución de cerrojo múltiple con una carga paramétrica del modelo de $P_S = 1$, $P_P = 0{,}8$, $P_M = 0{,}2$, $NP = 5$ y $\rho = 2$, sobre un tejido cuadrado de 8000 celdas de lado, durante 1000 pasos de tiempo discreto. Se realizan a partir de una única semilla *stem* en el centro del tejido. La máquina dispone de un procesador *quad-core* @3GHz y 8GB de RAM. Puede ver el resto de características en el apéndice B.

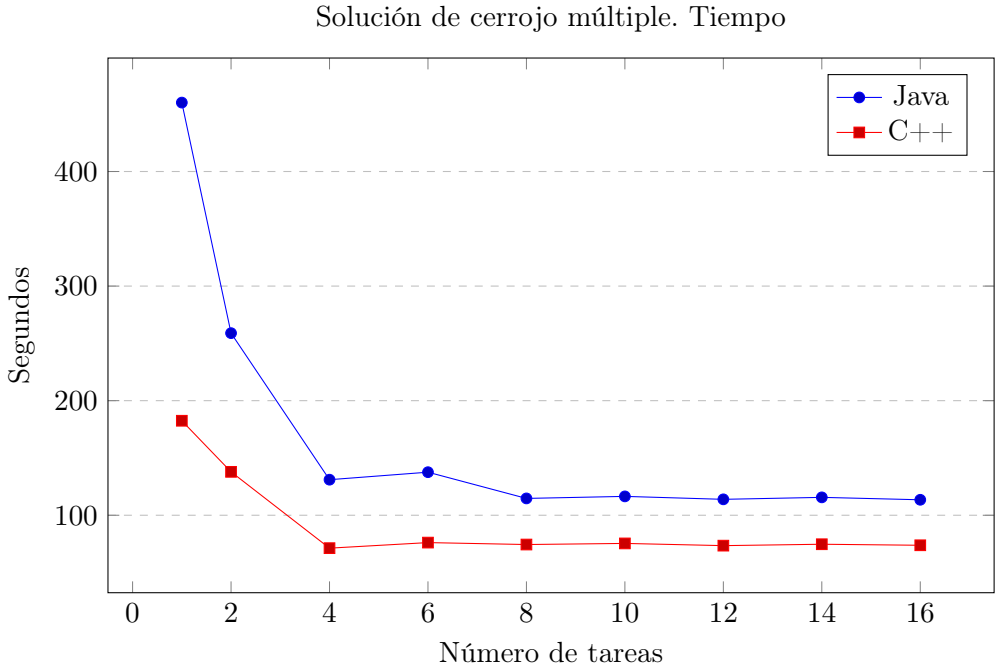

Figura 4.10: Tiempos de la solución de cerrojo múltiple con una carga paramétrica del modelo de $P_S = 1$, $P_P = 0{,}8$, $P_M = 0{,}2$, $NP = 5$ y $\rho = 2$, sobre un tejido cuadrado de 8000 células de lado, durante 1000 pasos de tiempo discreto. Se realizan a partir de una única semilla *stem* en el centro del tejido. La máquina dispone de un procesador *quad-core* @3GHz y 8GB de RAM. Puede ver el resto de características en el apéndice B.

rango de la vecindad de la célula más cercana. Planteamos, por ello, una solución que contemple únicamente las regiones conflictivas.

4.5.3. Cerrojo en Regiones de Conflicto

Estamos en situación de poder asegurar que no se precisa de un bloqueo del conjunto del sistema paralelo para procesar un autómata celular como el que presentamos. Es necesario conservar la integridad de la memoria, pero hay que hacerlo intentando no sacrificar el nivel de paralelismo.

Es posible tener consistencia en el dominio tisular bloqueando solo las regiones de conflicto. Denominamos como región de conflicto a los segmentos de datos de exactamente 4 celdas de anchura centrados en la frontera que limitan las particiones del tejido. Discernir por qué la anchura ha de ser cuatro no resulta tedioso: la acción de una célula $x_{i,j}$ dependerá de ciertas distribuciones de probabilidad y de la existencia de huecos en su vecindad. Tomada cualquier celda aledaña, por ejemplo, $x_{i-1,j}$, puede concluirse que el estado de dicha posición del tejido sólo podrá ser modificada por las células de alrededor. En concreto, y para el caso que estamos exponiendo, la célula más alejada a nuestra célula inicial, $x_{i,j}$, que puede cambiar su decisión, es la célula $x_{i-2,j}$. Por ello, para modificar la célula $x_{i,j}$ de la frontera, será necesario bloquear al otro hilo si se encuentra realizando cambios en un rango de dos posiciones más allá.

Realizando una simetría desde el punto de vista del otro hilo, e intersecando ambas regiones de conflicto, obtenemos un conjunto de 4 celdas de anchura.

Se asocia un cerrojo a las zonas fronterizas. Cada vez que un hilo necesite realizar cambios en la frontera, deberá capturar el cerrojo, debiendo esperar si otro hilo lo tiene capturado en ese momento.

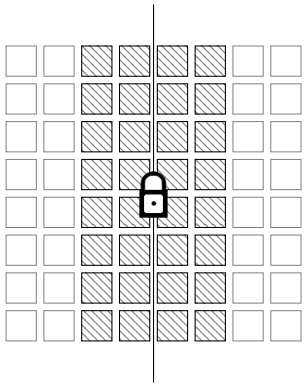

Figura 4.11: Representación de la aproximación
con un cerrojo por cada zona fronteriza

Se consigue de esta forma reducir significativamente la cantidad de bloqueos tomados por la aproximación de cerrojo múltiple.

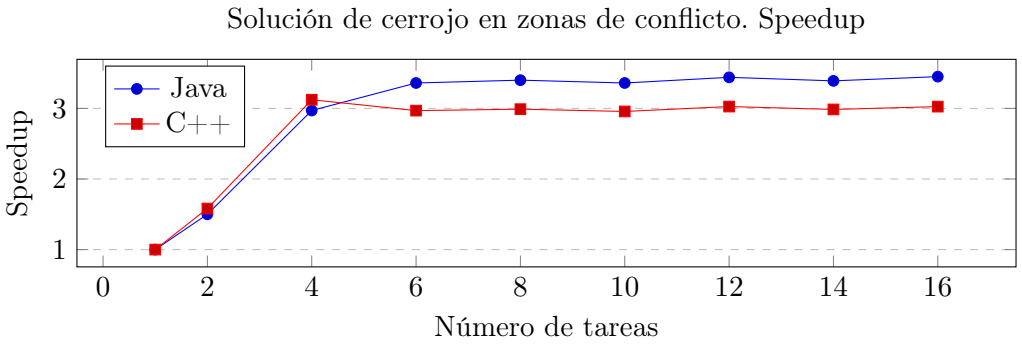

Figura 4.12: Speedup de la solución de cerrojo para fronteras con una carga paramétrica del modelo de $P_S = 1$, $P_P = 0{,}8$, $P_M = 0{,}2$, $NP = 5$ y $\rho = 2$, sobre un tejido cuadrado de 8000 celdas de lado, durante 1000 pasos de tiempo discreto. Se realizan a partir de una única semilla *stem* en el centro del tejido. La máquina dispone de un procesador *quad-core* @3GHz y 8GB de RAM. Puede ver el resto de características en el apéndice B.

Número de tareas	Tiempo Java (seg.)	Tiempo C++ (seg.)
1	460.15	354.47
2	306.57	224.08
4	154.81	113.52
6	136.84	119.41
8	135.13	118.59
10	136.89	119.88
12	133.65	117.14
14	135.63	118.74
16	133.28	117.17

Tabla 4.4: Ejecución de la solución de cerrojo en regiones de conflicto con una carga paramétrica del modelo de $P_S = 1$, $P_P = 0{,}8$, $P_M = 0{,}2$, $NP = 5$ y $\rho = 2$, sobre un tejido cuadrado de 8000 celdas de lado, durante 1000 pasos de tiempo discreto. Se realizan a partir de una única semilla *stem* en el centro del tejido. La máquina dispone de un procesador *quad-core* @3GHz y 8GB de RAM. Puede ver el resto de características en el apéndice B.

No hemos conseguido una mejora significativa respecto a las versiones anteriores. De hecho, en algunos casos puede observarse un empeoramiento en los tiempos de ejecución, debido a la introducción de lógica adicional para manejar los cerrojos. En el caso particular de C++, obtenemos peores tiempos durante todo el domino de pruebas.

Dado que el número de bloqueos se ha reducido al mínimo, el cuello de botella en la ejecución tiene que estar en otro sitio. Se nos plantean dos posibilidades: el generador de números aleatorios y el acceso a memoria.

Es lógico que generar una distribución de probabilidad tenga un coste asociado, más aun si se trata de aleatorios reales [35]. En el caso de Java, esto escapa a nuestro control; en el caso de C++, se ha usado `std::random_device`, que bajo sistemas ope-

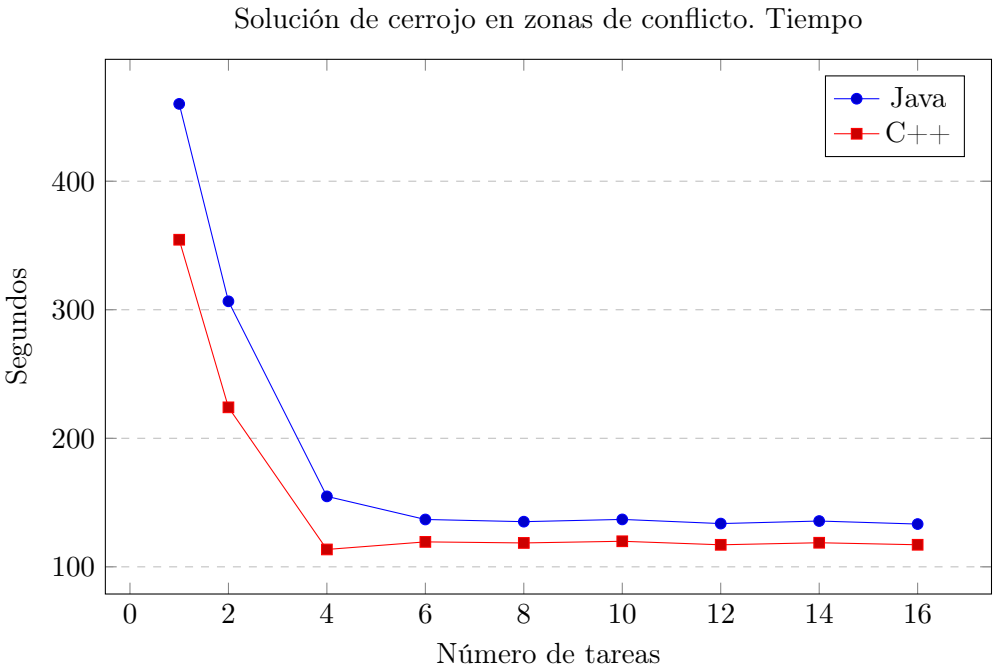

Figura 4.13: Tiempos de la solución de cerrojo para regiones de conflicto con una carga paramétrica del modelo de $P_S = 1$, $P_P = 0{,}8$, $P_M = 0{,}2$, $NP = 5$ y $\rho = 2$, sobre un tejido cuadrado de 8000 células de lado, durante 1000 pasos de tiempo discreto. Se realizan a partir de una única semilla *stem* en el centro del tejido. La máquina dispone de un procesador *quad-core* @3GHz y 8GB de RAM. Puede ver el resto de características en el apéndice B.

rativos basados en el kernel de Linux, hace uso del dispositivo virtual `/dev/random` (ver [35]).

La posible optimización de la generación de pseudoaleatrios no se ha investigado, y por tanto puede ser tenida en cuenta como trabajo futuro.

En el otro extremo, encontramos el acceso a memoria. Ya hemos hablado en la sección 4.1 de como optimizar las latencias haciendo uso del concepto de localidad espacial. A pesar de ello, es obvio que no es suficiente. Es necesario plantear otro enfoque que reduzca drásticamente los accesos. Se propone, por ello, implementar un autómata con dominio de cómputo creciente, que no procese células que no sean necesarias.

4.6. Dominio Creciente

Una vez reducidos los bloqueos necesarios para asegurar la integridad de la memoria, sigue siendo necesario disminuir los tiempos de ejecución.

El acceso a memoria es un cuello de botella nada despreciable. Como ya hemos comentado anteriormente, en las arquitecturas actuales se intenta reducir mediante un buen diseño de la jerarquía de memoria, colocando varios niveles de caché, normalmente asociativa (ya sea asociativa simple o por conjuntos).

Existen diferentes problemas inherentes al funcionamiento de las cachés, a parte del reducido tamaño, como el uso compartido falso (*false sharing*), derivados de

arquitecturas *multicore*. Ocurre cuando diferentes procesadores intentan acceder a regiones distintas de memoria y almacenar dicha región en la misma línea de caché, provocando la reescritura de datos que pueden ser usados por otro *core*. No es fácil solucionarlo, ya que depende de la organización de los datos en memoria y, normalmente, solventarlo pasa por realinearla.

Resulta más sencillo limitar los accesos a memoria. Por el comportamiento de nuestro algoritmo, sabemos que una célula muerta no puede realizar ninguna acción.

Un tumor sólido, como el cáncer de mama, comienza a ser detectable [36] a partir de unos 7 milímetros (de media) usando técnicas de imagen, y a partir de 15 milímetros comienza a ser palpable (exploración clínica). Considerando células tumorales de unos $10\mu m^2$, y a partir de la ecuación presentada en [37], hemos calculado la necesidad de un dominio tisular de tamaño, como mínimo, 10^8 células, si queremos que sea detectable por imagen.

El tiempo que un tumor tarda en alcanzar dicho tamaño oscila entre 2 y 8 años, dependiendo de diferentes parámetros. Esto es, entre 17500 y 70080 pasos de tiempo discreto. Es decir, si instanciamos un dominio tisular de 10^8 células, tardaremos entre 17500 y 70080 iteraciones en ocupar toda la matriz. ¿Tiene sentido, por tanto, procesar posiciones desocupadas de la matriz durante todas esas iteraciones?

Obviamente la respuesta es no. Lo hemos solucionado mediante la introducción en el algoritmo de un dominio de procesamiento creciente, de tal forma que no se incluyen en la ejecución células que se sabe que están muertas. No se particiona toda la matriz, sino única y exclusivamente las células que se encuentran dentro del dominio de cómputo.

El dominio de cómputo es un rectángulo definido por dos coordenadas, que corresponden a su esquina superior izquierda y su esquina inferior derecha. Dichas coordenadas se actualizan en cada iteración, y corresponden a las células situadas más arriba y a la izquierda y más abajo y a la derecha, respectivamente.

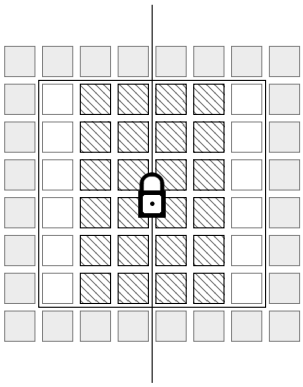

Figura 4.14: Representación de la aproximación con un cerrojo por cada zona fronteriza, con dominio de cómputo creciente

Se consigue así independizar el tiempo de ejecución del tamaño máximo del dominio tisular, haciéndolo depender únicamente del número de pasos discretos.

Los resultados son espectaculares. Hemos obtenido mejoras en tiempo muy significativas (tabla 4.5, figuras 4.15 y 4.16). Como ejemplo, para el dominio tisular de

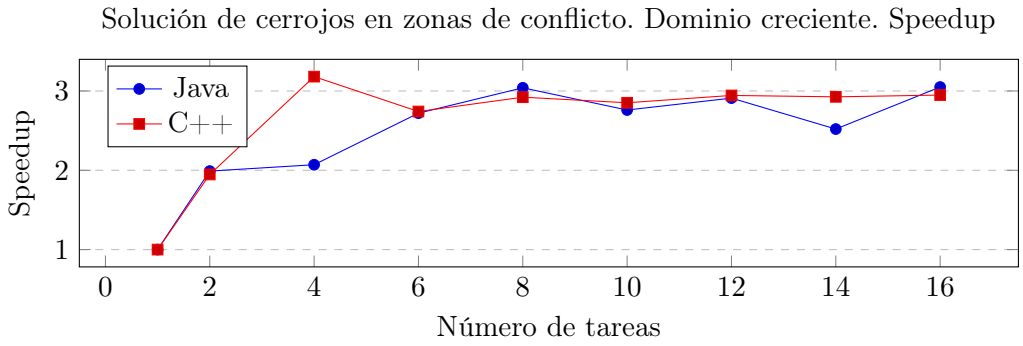

Figura 4.15: Speedup de la solución de cerrojos para fronteras, con dominio de ejecución creciente y una carga paramétrica del modelo de $P_S = 1$, $P_P = 0{,}8$, $P_M = 0{,}2$, $NP = 5$ y $\rho = 2$, sobre un tejido cuadrado de 8000 celdas de lado, durante 4000 pasos de tiempo discreto. Se realizan a partir de una única semilla *stem* en el centro del tejido. La máquina dispone de un procesador *quad-core* @3GHz y 8GB de RAM. Puede ver el resto de características en el apéndice B.

8000^2 células, hemos conseguido una mejora de un 6077 %, teniendo en cuenta los dos mejores tiempos de cada implementación.

Número de tareas	Tiempo Java (seg.)	Tiempo C++ (seg.)
1	3.34	2.95
2	2.46	1.67
4	2.29	1.08
6	2.06	1.49
8	1.95	1.34
10	1.92	1.52
12	1.87	1.43
14	1.87	1.51
16	1.88	1.48

Tabla 4.5: Ejecución de la solución de cerrojos en regiones de conflicto y dominio de ejecución creciente, con una carga paramétrica del modelo de $P_S = 1$, $P_P = 0{,}8$, $P_M = 0{,}2$, $NP = 5$ y $\rho = 2$, sobre un tejido cuadrado de 8000 celdas de lado, durante 1000 pasos de tiempo discreto. Se realizan a partir de una única semilla *stem* en el centro del tejido. La máquina dispone de un procesador *quad-core* @3GHz y 8GB de RAM. Puede ver el resto de características en el apéndice B.

El comportamiento observado resulta bastante curioso. Se han reducido los tiempos drásticamente, pero ya no existe apenas diferencia entre Java y C++. De hecho, el primero supera al segundo en ciertos casos. A lo largo de todas las pruebas comprobadas hemos observado que el rendimiento de C++ es mejor cuando el número de tareas lanzadas es igual al número de núcleos disponible. En Java, por lo general, los mejores resultados se han obtenido con un número de tareas superior a los *cores* de la máquina.

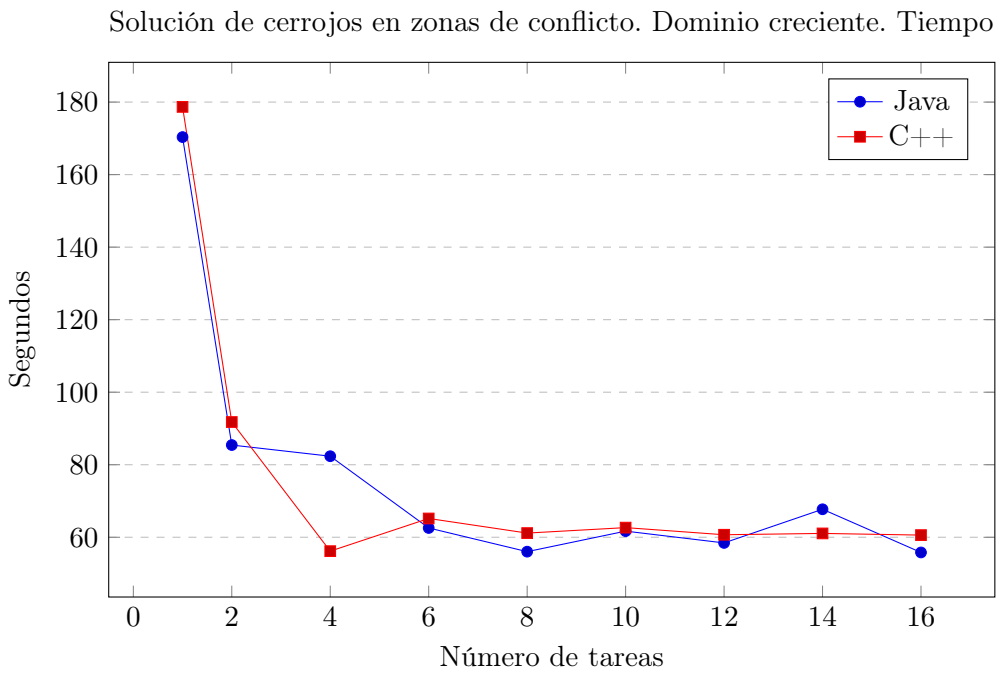

Figura 4.16: Tiempos de la solución de cerrojos para regiones de conflicto, con dominio de ejecución creciente y una carga paramétrica del modelo de $P_S = 1$, $P_P = 0{,}8$, $P_M = 0{,}2$, $NP = 5$ y $\rho = 2$, sobre un tejido cuadrado de 8000^2 células, durante 4000 pasos de tiempo discreto. Se realizan a partir de una única semilla *stem* en el centro del tejido. La máquina dispone de un procesador *quad-core* @3GHz y 8GB de RAM. Puede ver el resto de características en el apéndice B.

4.7. Ejecución en Clúster de Apoyo a la Investigación

Para todas las versiones en Java descritas en este capítulo, hemos llevado a cabo pruebas en el clúster de apoyo a la investigación de la Universidad de Cádiz. Se han usado 4 nodos para ofrecer datos estadísticos básicos (media y desviación típica) y se han presentado mediante las gráficas y tablas que el lector puede analizar en los apéndices C y D.

No se han llevado a cabo pruebas en C++ debido a limitaciones técnicas ajenas al autor. El clúster dispone de una versión antigua del compilador de C++ de GNU sin soporte completo al estándar de 2011.

Capítulo 5

Conclusiones y Trabajos Futuros

Ya hemos hablado en el punto 1.2 de los avances actuales en materia de simuladores paralelos de neoplasias.

A lo largo de este trabajo se han establecido diferentes aproximaciones para paralelizar nuestro autómata celular, se han precisados tiempos de ejecución en máquinas domésticas, así como índices de aceleración del modelo paralelo.

5.1. Conclusiones

De todo lo anterior, podemos extraer las siguientes conclusiones:

- Los experimentos *in silico* son capaces de realizar simulaciones significativas en poco más de una hora, usando máquinas de cómputo domésticas.
- Hay que tener en cuenta ciertos aspectos, como la dirección y sentido de iteración, si queremos ajustar el modelo a la realidad.
- Los autómatas celulares proporcionan una gran flexibilidad, en tanto que la regla del autómata puede ser definida tan compleja como se quiera.
- Las distribuciones de probabilidad pueden ajustarse de tal forma que se puedan simular escenarios difíciles de alcanzar en la realidad, por limitaciones físicas o costes elevados.
- Se puede obtener una simulación aceptable usando aproximaciones secuenciales (en máquina doméstica, una simulación de dos años puede hacerse en unas 4 horas, con dominio de cómputo creciente), pero estaremos desaprovechando gran parte de la capacidad de los sistemas de computación actuales.
- Paralelizar sólo requiere controlar la sincronización de los hilos (esperar a que todos acaben de iterar para continuar con la siguiente generación) y garantizar la consistencia de la memoria.
- Por ello, es fácil introducir mejoras paralelas (un simple bloqueo a todo el dominio tisular) para obtener aceleraciones aceptables. Creemos que cualquier investigador con conocimientos de programación puede realizarlo.

- Tomando los bloqueos de forma más granular se consiguen aceleraciones decentes. La aproximación que los minimiza solo requiere de la introducción de un entero (índice del hilo), los cerrojos necesarios y una bifurcación (que decida si se está en una zona donde puede haber inconsistencia y bloquee en consecuencia). Ésta debe ser la aproximación elegida; no requiere demasiados ajustes en el código y creemos que puede ser implementada por cualquier investigador con conocimientos de programación.

- Es preciso disminuir los accesos a memorial. El dominio de cómputo creciente es la mejora más notable de todas las introducidas.

- Los lenguajes ejecutados en máquina virtual no necesariamente son más lentos que los lenguajes compilados a código máquina. A lo largo de este trabajo, hemos podido observar como C++ mejora el rendimiento secuencial de Java (probablemente debido a la gran cantidad de optimizaciones) pero su *speedup* suele ser menor. En simulaciones reales (véanse las figuras B.18, B.20, B.30, B.31 y B.32) ambos lenguajes se comportan de forma muy similar en tiempo. Java incluso mejora algo a C++.

5.2. Trabajos Futuros

Es posible seguir, y de hecho se debe seguir, mejorando estos simuladores mediante autómatas celulares. Entre nuestros trabajos futuros se encuentran:

- Ajustar el modelo para representar con aun más fidelidad la realidad de las neoplasias. Contemplar características como la angiogénesis o simular el efecto de las drogas existentes o en proceso de estudio contra el cáncer puede resultar de infinita utilidad para la comunidad investigadora.

- Contemplar y analizar las distintas formas de generación de aleatorios y pseudoaleatorios, así como las distintas distribuciones de probabilidad (por ejemplo, las incluidas en la biblioteca estándar de C++), desde un punto de vista del rendimiento y de la adaptabilidad a la simulación tumoral.

- Las arquitecturas *manycore* se han hecho un hueco con fuerza en el mundo del cálculo intensivo y se incluyen en muchos *clusters* de apoyo a la investigación. La aceleración hardware mediante *grids* de tarjetas gráficas (GPU) con tecnologías como CUDA™ pueden resultar en una mejora muy significativa del rendimiento.

- De la misma forma, existen lenguajes de programación modernos como *Go* y *C#* y APIs orientadas a la programación multiproceso y distribuida como *OpenMP* que deberían ser valorados como alternativas a los aquí presentados.

- Evaluar el rendimiento de tecnologías como GCD y STM, mediante técnicas libres de bloqueo que garanticen la consistencia de la memoria.

- Teselar el tejido (matriz del autómata celular) implica segmentos de datos perfectamente *cacheables*. Realizando un paralelismo de datos de grano algo más fino (por ejemplo, submatrices de 64x64 células) se aproxima el modelo a la forma de almacenaje y tratamiento de imágenes que se usan en la actualidad,

y pueden resultar en rendimientos mejorados (a pesar de que, probablemente, se necesite elevar el número de bloqueos al realizar operaciones de vecindad).

Capítulo 6

Resolución de Conflictos (*Troubleshooting*)

Se presentan algunos de los problemas con los que nos hemos encontrado en la elaboración de este trabajo y que hemos considerado relevantes. Quedan resumidos y recogidos a continuación.

6.1. Sentido de Iteración

Calcular una generación de un autómata celular implica recorrerlo. Los bucles anidados siempre recorren en el mismo sentido, por lo que los cambios en la generación no se producen espontáneamente, todos a la vez, sino de forma ordenada.

El principal inconveniente de esto es la deformidad que se produce si no se toman medidas para paliarla. Una célula cualquiera puede proliferar a una celda libre en la vecindad. Si la celda ya ha sido iterada previamente, no ocurrirá nada. Si la célula va a ser iterada posteriormente, es posible que realice una acción, como migrar o proliferar. Con el suficiente número de ocurrencias de esta situación, obtenemos tumores sólidos muy deformes.

Esto ha sido solucionado mediante la introducción de una matriz de *banderas* (`generations_`) que especifica qué células han de ser iteradas y qué células no.

A pesar de esto, pueden seguir apreciándose deformidades si se producen migraciones en bloque. Esto es, cuando una célula decide migrar hacia una posición ya iterada y las siguientes se van moviendo *al hueco*. Hemos preferido no solventar este tipo de migraciones, puesto que ocurren en la naturaleza: una célula puede hacerse hueco desplazando aquellas que están a su alrededor y, a efectos prácticos, este comportamiento implica lo mismo.

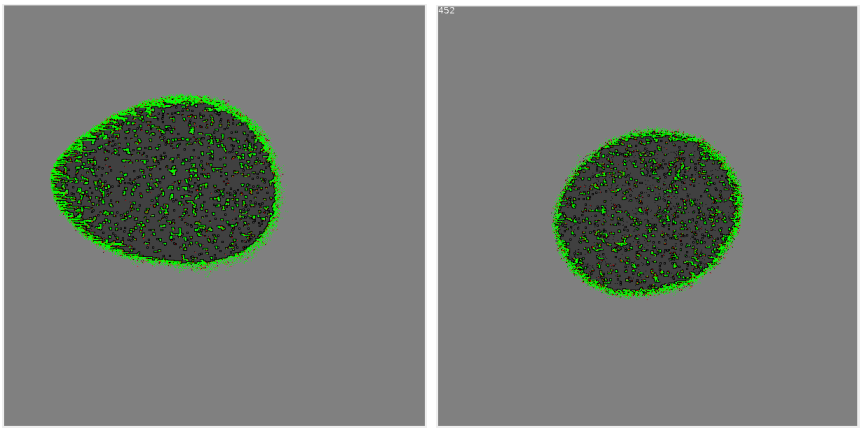

Figura 6.1: A la izquierda, el problema sin solucionar. A la derecha, la versión con diferentes sentidos de iteración. Ambas han sido ejecutadas con los mismos parámetros. las células verdes representan migraciones; las rojas, proliferaciones.

Lo que sí hemos resuelto es la asimetría. Estas migraciones han de hacerse en ambos sentidos. La forma de arreglarlo ha pasado por variar el sentido de iteración con cada generación. De esta forma, se itera en distinto orden en las generaciones impares, concretamente, en el sentido opuesto a las pares. Los desplazamientos se producen equiprobablemente hacia ambos lados, por lo que la neoplasia permanece simétrica.

6.2. Sincronización en C++

Ya hemos comentado anteriormente que los recursos disponibles en ámbito de concurrencia en C++ son limitados, y que se espera que sigan aumentando en las próximas versiones del lenguaje. Una de las cosas de las que precisamos es de un método de sincronización (barrera) que detenga a las diferentes hebras hasta que todas hayan acabado de iterar para continuar con el cálculo de la siguiente generación

Hemos realizado una implementación usando objetos `std::condition_variable` (variables de condición con un funcionamiento similar a las que proporciona Java). Una solución altamente optimizada incluida en la biblioteca estándar de C++ podría suponer una ventaja en términos de rendimiento.

Apéndice A

Análisis de la Entropía

En el apartado 2.5 hemos definido la entropía y descrito algunas de sus propiedades. En este apéndice, queremos estudiar el comportamiento de la curva que ofrece este índice para escenarios conocidos.

A.1. Tumores Densos

Los tumores densos se caracterizan por concentrar todas o gran parte de las células cancerosas en una única masa surgida en torno a las células *stem* iniciales.

La curva de entropía de estos tumores está, según nuestras pruebas, bastante bien definida. Se observan tres estados básicos (figura A.1):

- Crecimiento. La curva sigue una monotonía creciente hasta que alcanza su máximo absoluto. Se trata del momento de máxima expansión de la neoplasia. Conforme va creciendo el número de células latentes y vivas y decreciendo la cantidad de células muertas, el grado de incertidumbre aumenta. La curva llega a su máximo absoluto cuándo el número de células muertas, latentes y vivas es aproximadamente igual.

- Disminución. La curva sigue una monotonía decreciente. Corresponde con la fase opuesta a la anterior: el número de células vivas y latentes aumenta desmesuradamente, saturando el dominio tisular y, por consiguiente, disminuyendo las probabilidades de que, tomada una célula al azar, ésta esté muerta. El grado de incertidumbre decrece, y con él, la curva de entropía.

- Estabilización. Una vez el dominio tisular está saturado, la mayor parte de células estarán vivas o latentes. Esto hace que la incertidumbre se estabilice en torno a una constante. Las fluctuaciones observables son despreciables, y se corresponden con variaciones en la cantidad de células en cada estado. Si la probabilidad de supervivencia es 1, eventualmente todas las células se quedarán sin espacio en la vecindad, de tal forma que pasarán a estar latentes. Si éste es el caso, el grado de incertidumbre será nulo, pues si realizamos una extracción aleatoria, la probabilidad de que el estado de la célula sea latente será de 1. En cualquier otro caso, la entropía nunca será nula (salvo apoptosis de todas las células).

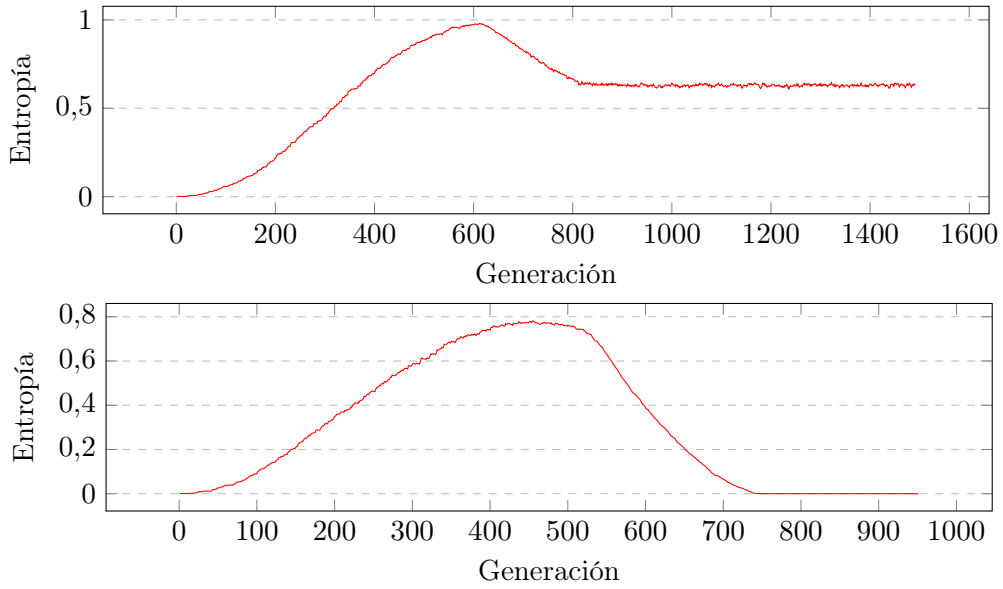

Figura A.1: Gráficas de entropía para un tumor denso de 200^2 células. Pueden observarse las tres fases descritas. La gráfica superior se corresponde con una simulación cuya carga paramétrica es $P_S = 0{,}99$, $P_P = 0{,}4$, $P_M = 0{,}2$, $NP = 5$ y $\rho = 2$. La gráfica inferior se ha ejecutado con los mismos parámetros, salvo $P_S = 1$.

A.2. Tumores Dispersos

Los tumores dispersos se caracterizan por no tener todas las células concentradas en una única masa. Las semillas *stem* se reparten por todo el dominio tisular, de forma uniforme.

Describen el mismo comportamiento que los sólidos. La curva denota fases idénticas a las analizadas para tumores sólidos, con la única diferencia de que la fase de crecimiento es más corta: al existir varias masas, el crecimiento es más rápido, no hay tantas células latentes y, por tanto, el máximo se alcanza antes. Puede verse este comportamiento en la figura A.2.

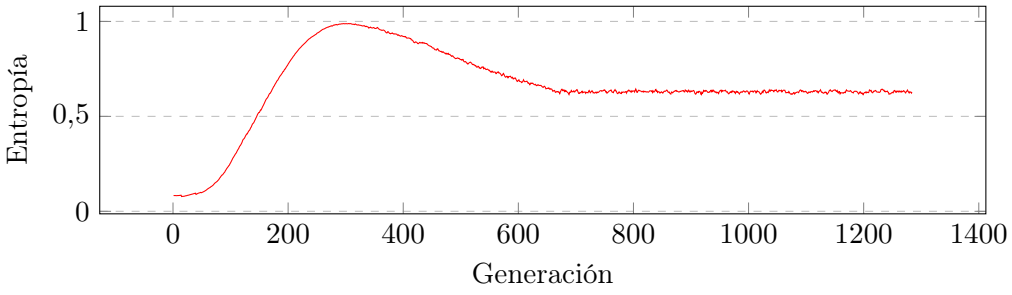

Figura A.2: Gráficas de entropía para un tumor disperso de 200^2 células. Pueden observarse las tres fases descritas y la corta duración de la fase de crecimiento. La gráfica se corresponde con una simulación cuya carga paramétrica es $P_S = 0{,}99$, $P_P = 0{,}4$, $P_M = 0{,}2$, $NP = 5$ y $\rho = 2$. Se han establecido 15 semillas iniciales con distribución aleatoria uniforme por todo el dominio tisular.

A.3. Parámetros Extremos

Establecer valores extremos en los diferentes parámetros no modifica el comportamiento de la curva (salvo establecer la probabilidad de supervivencia a 0, lo cual anula la incertidumbre), aunque sí su velocidad de crecimiento, el valor de su máximo, la velocidad de disminución y la constante entorno a la que se oscila al estabilizarse.

Por lo general, cuanto más parecidos son los parámetros que regulan la simulación, mayor es la entropía y más suave es el perfil descrito por la entropía.

Apéndice B

Medidas en Máquina Doméstica

Se incluyen a continuación las gráficas de tiempo y *speedup* (no se usa el mejor algoritmo como numerador, sino la versión de una única tarea de cada uno). La carga paramétrica de todas las simulaciones se corresponde con $P_S = 1$, $P_P = 0{,}8$, $P_M = 0{,}2$, $NP = 5$ y $\rho = 2$. En las simulaciones de domino computacional estático se varía el tamaño del dominio tisular y se mantienen 1000 generaciones. En las simulaciones de domino computacional dinámico se mantiene un tamaño de 8000^2 células y se varía el número de iteraciones.

Las pruebas presentadas se realizan en Java, versión 1.8.0_77 de la OpenJDK, y en C++, compilador *g++* versión 5.3.0, estándar C++11, nivel 3 de optimización, sobre ArchLinux (64 bits) bajo el kernel 4.4.5-1 de linux. La máquina dispone de un procesador AMD Athlon$^{\text{TM}}$ II X4 640 @3GHz, *quad-core*, 2 MB de caché L2, 512 KB de caché L1, sin caché L3, junto a 8GB de memoria RAM DDR2-1200 *dual-channel*.

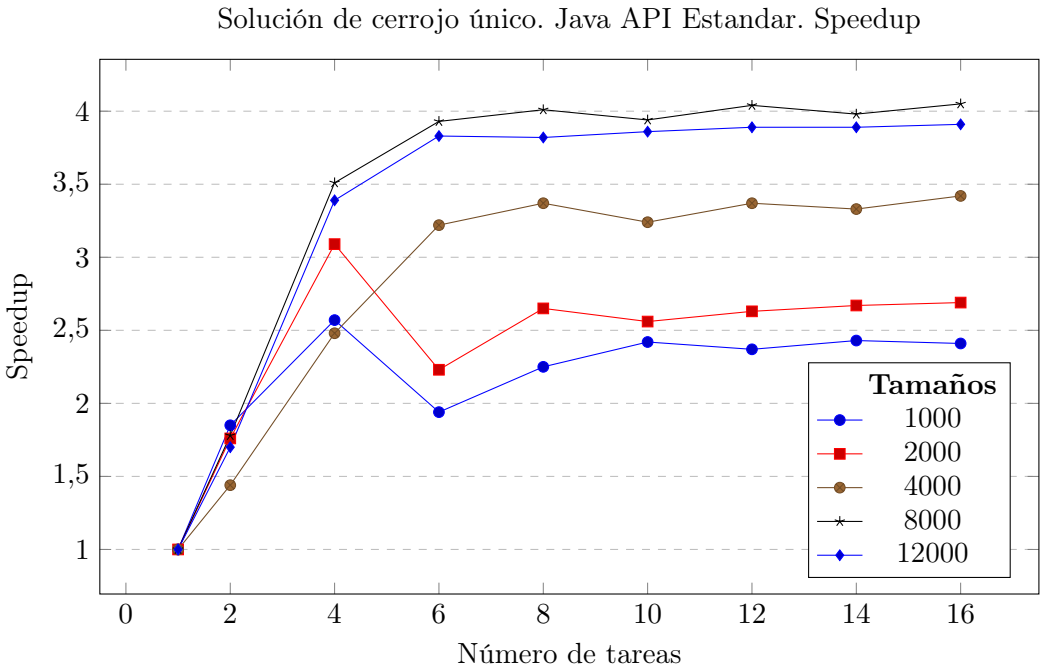

Figura B.1: Speedup de la ejecución de la solución de cerrojo único en Java, API Estándar.

ss

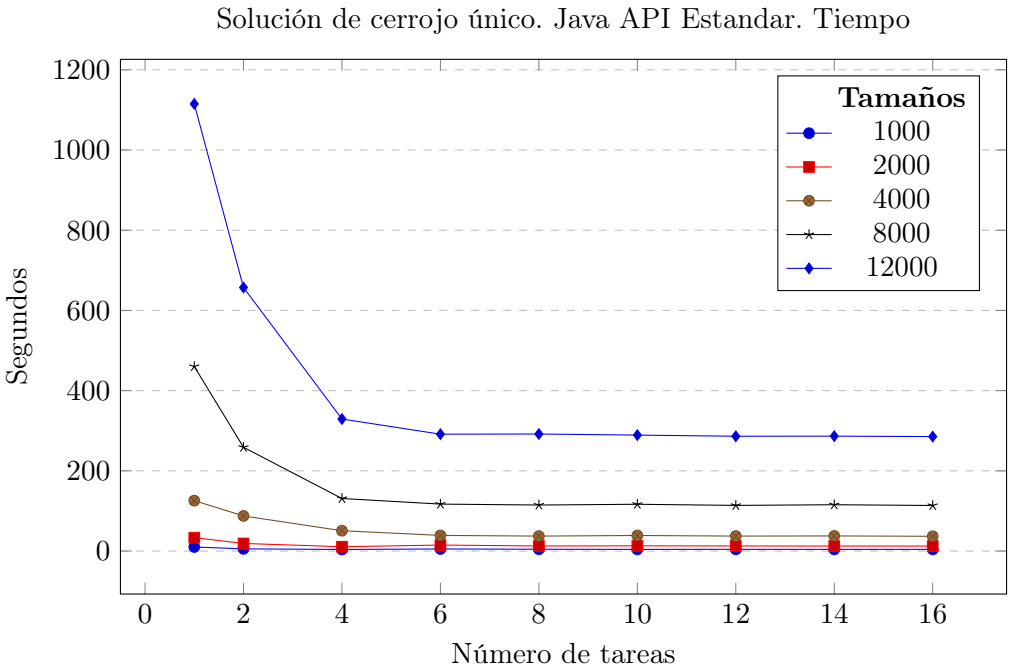

Figura B.2: Tiempos de la ejecución de la solución de cerrojo único en Java, API Estándar.

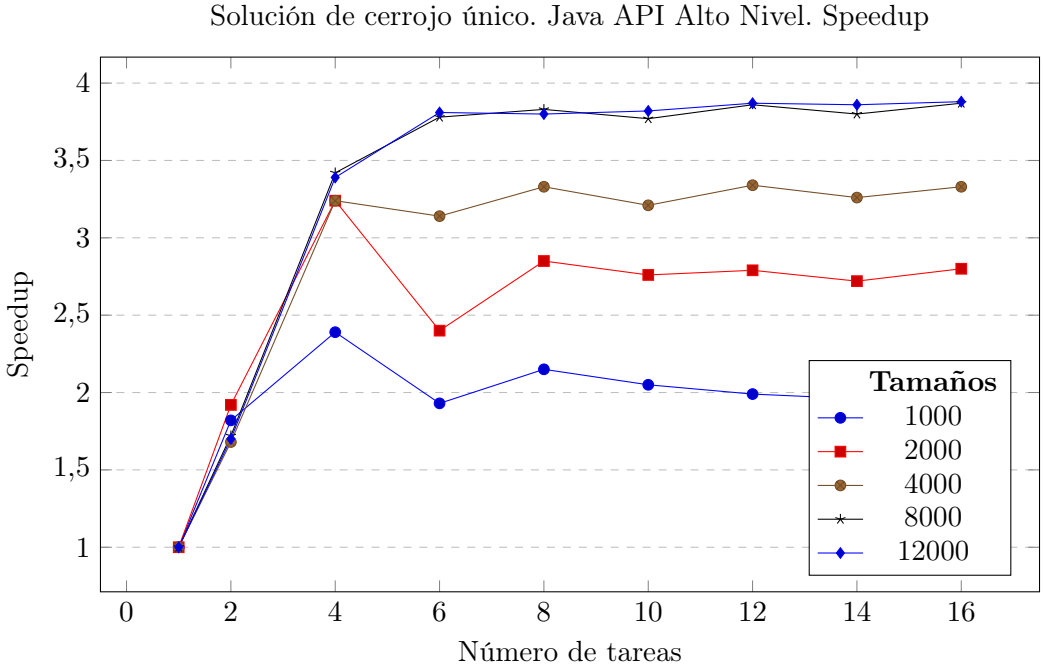

Figura B.3: Speedup de la ejecución de la solución de cerrojo único en Java, API Alto Nivel.

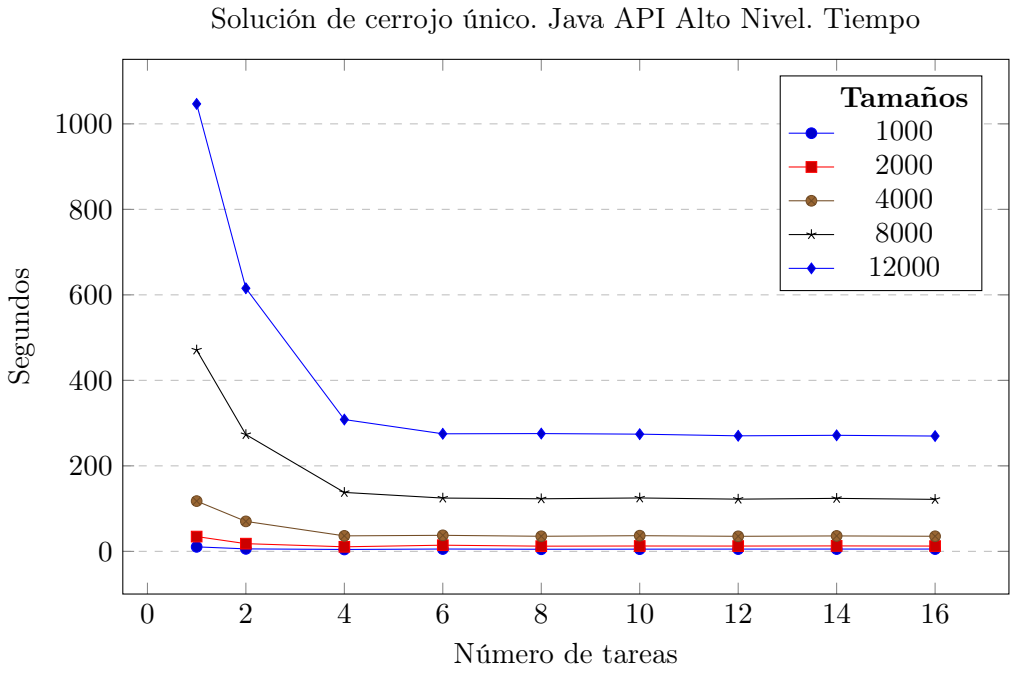

Figura B.4: Tiempos de la ejecución de la solución de cerrojo único en Java, API Alto Nivel.

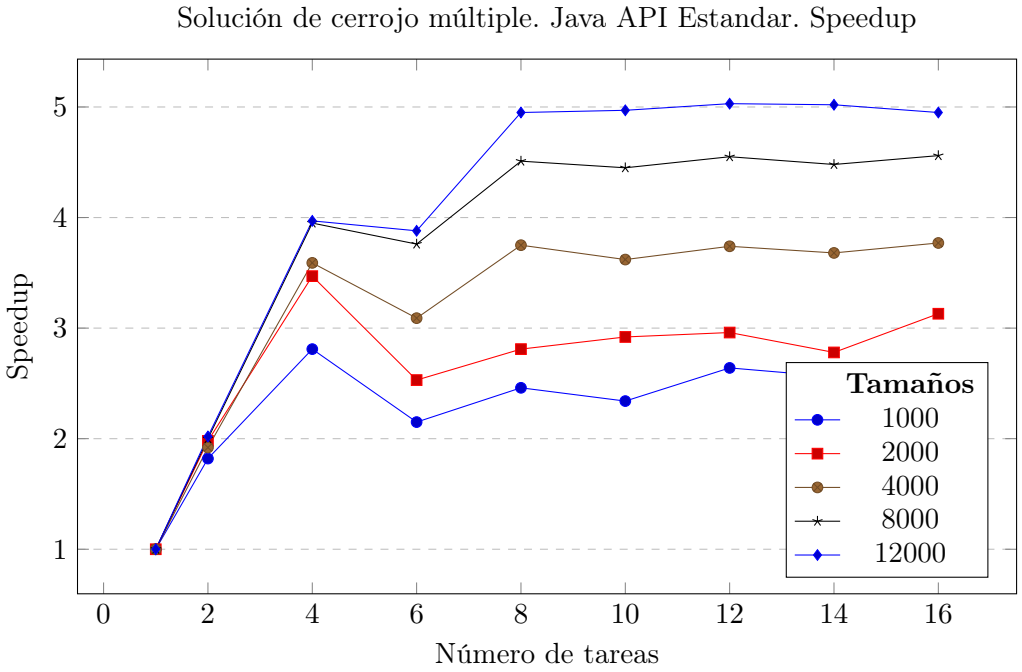

Figura B.5: Speedup de la ejecución de la solución de un cerrojo por partición en Java, API Estandar.

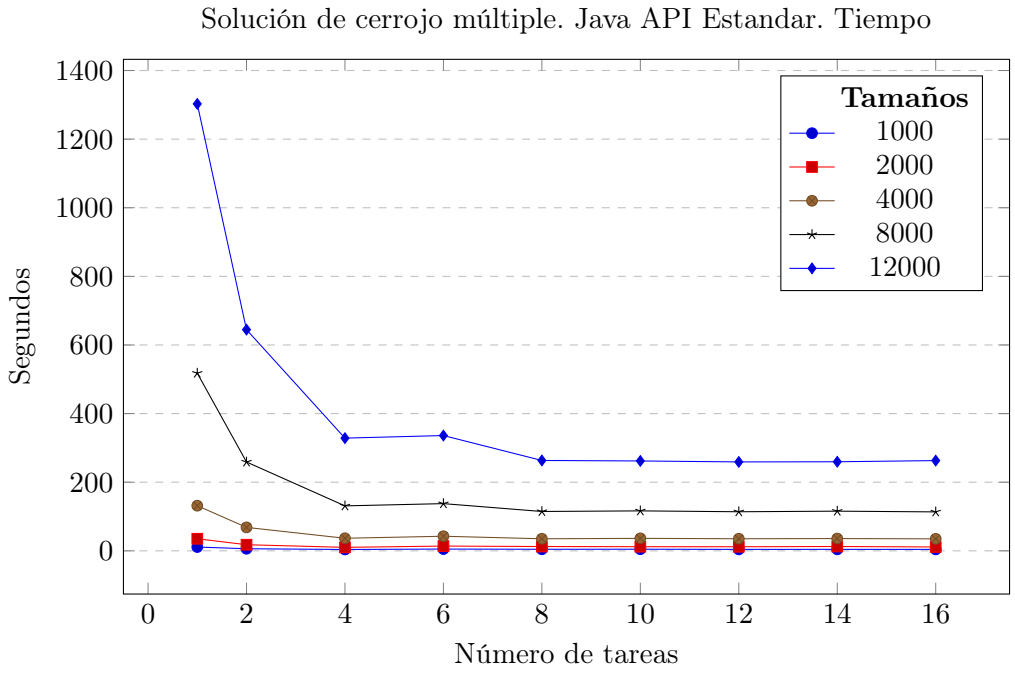

Figura B.6: Tiempos de la ejecución de la solución de un cerrojo por partición en Java, API Estandar.

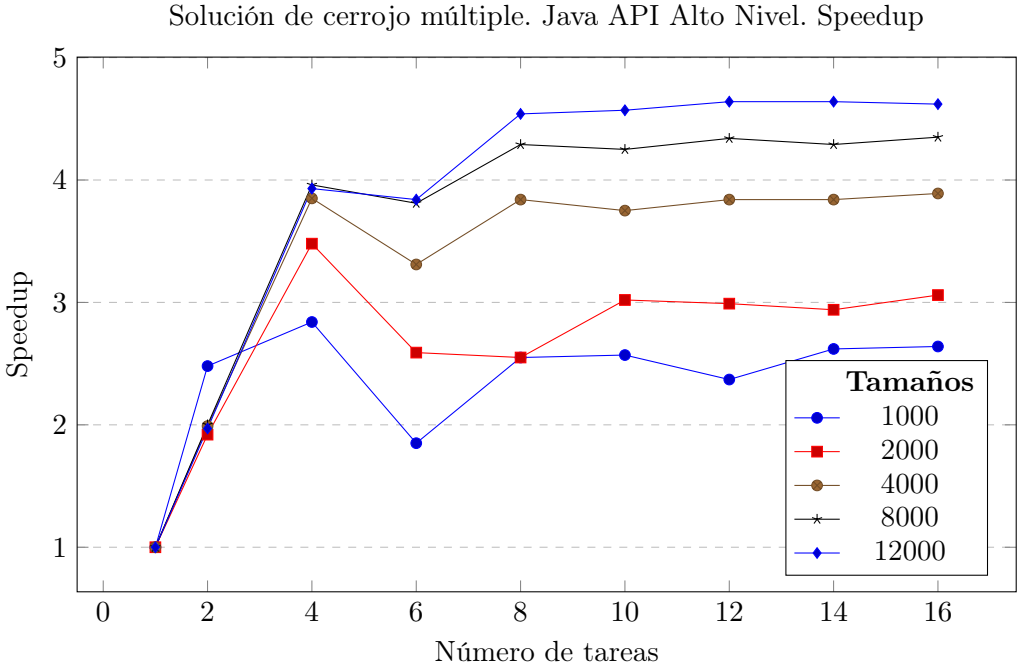

Figura B.7: Speedup de la ejecución de la solución de un cerrojo por partición en Java, API Alto Nivel.

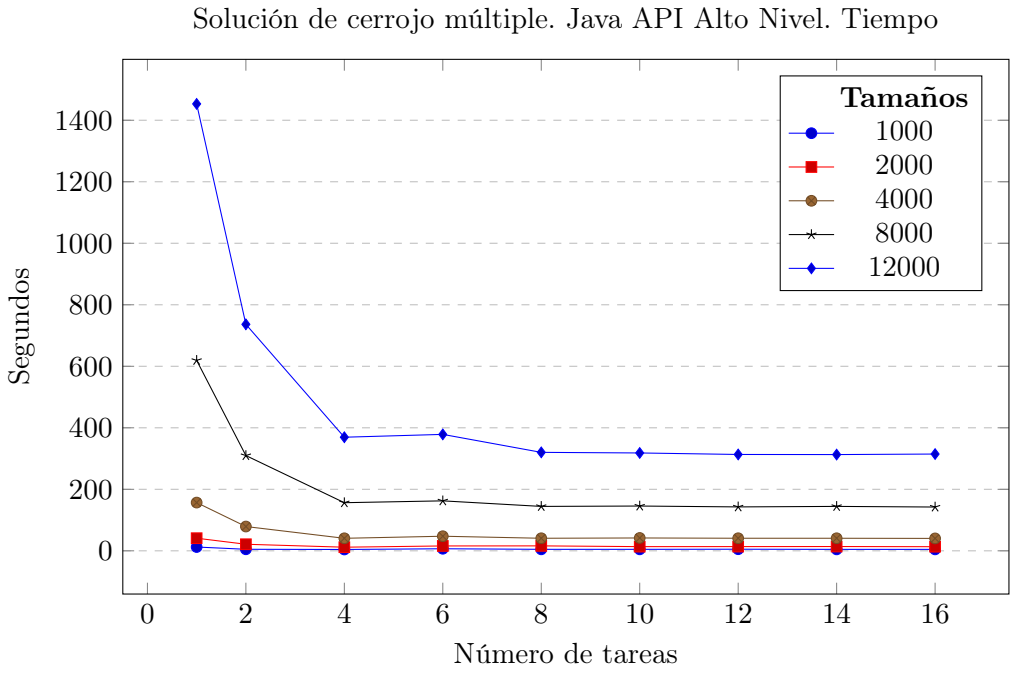

Figura B.8: Tiempos de la ejecución de la solución de un cerrojo por partición en Java, API Alto Nivel.

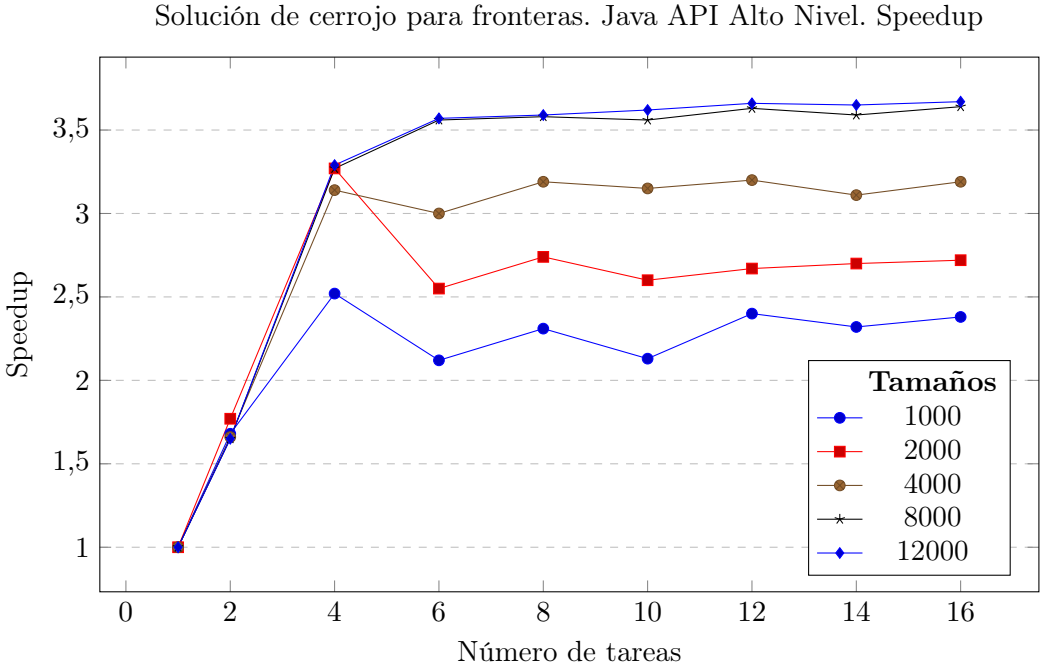

Figura B.9: Speedup de la ejecución de la solución de cerrojo en zona fronteriza en Java, API Alto Nivel.

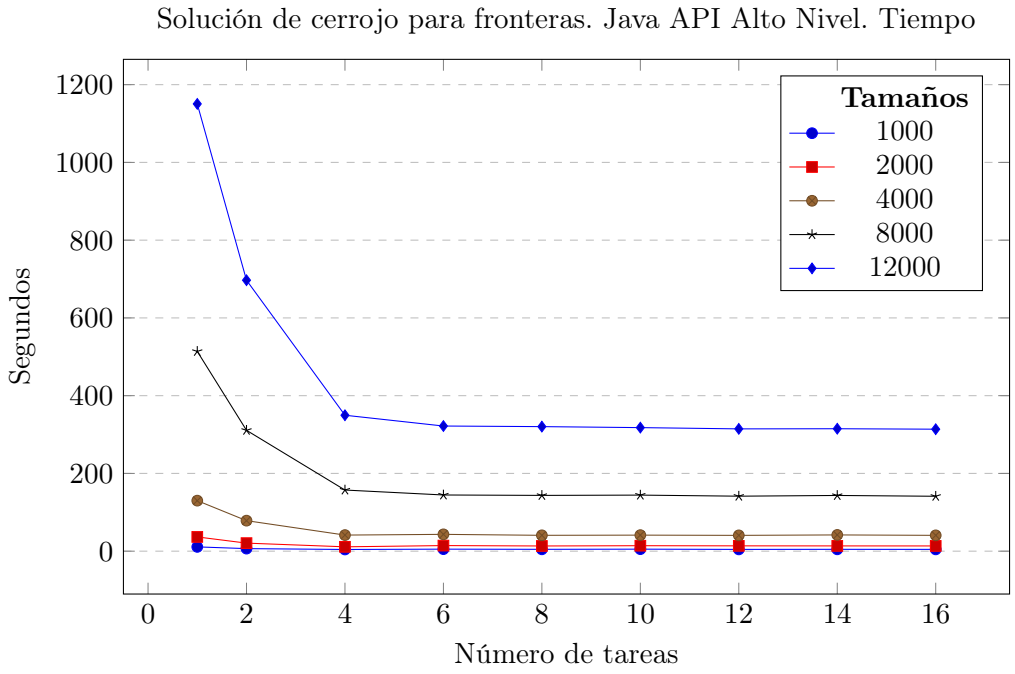

Figura B.10: Tiempos de la ejecución de la solución de cerrojo en zona fronteriza en Java, API Alto Nivel.

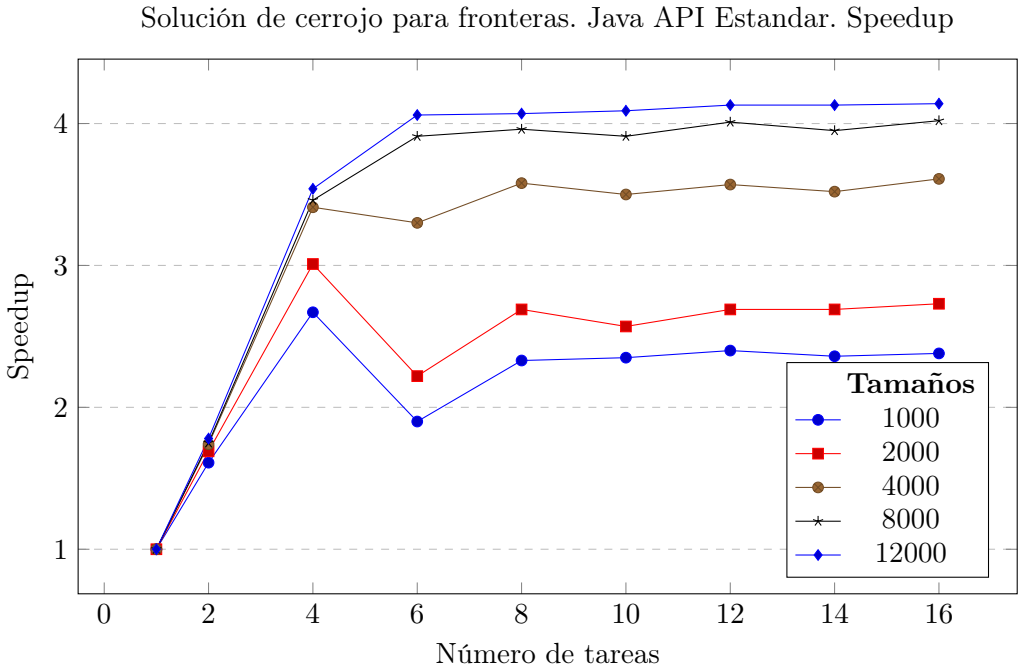

Figura B.11: Speedup de la ejecución de la solución de cerrojo en zona fronteriza en Java, API Estándar.

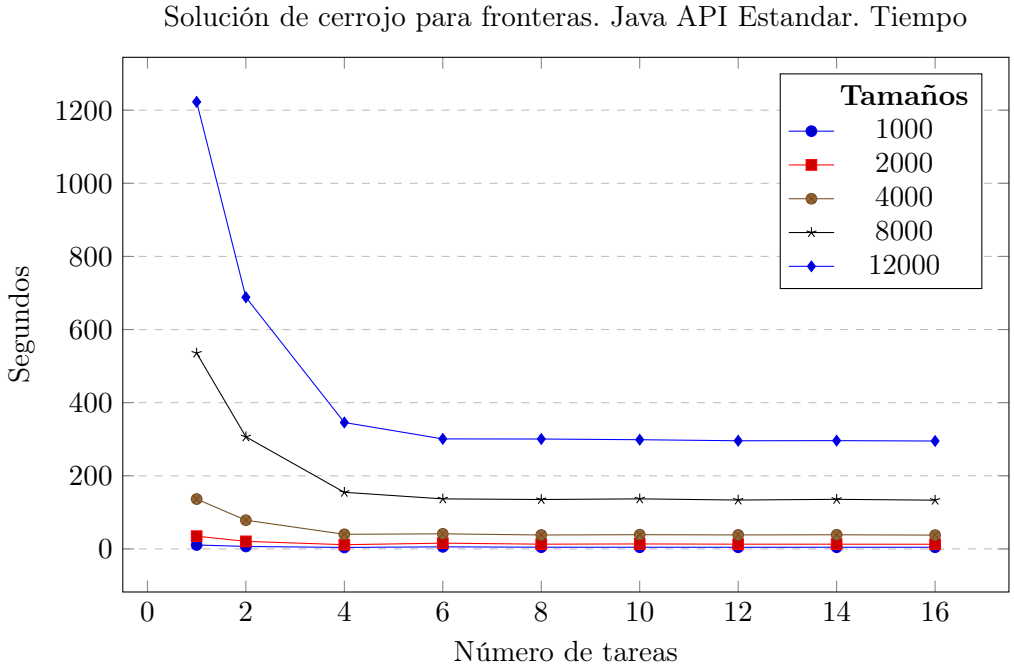

Figura B.12: Tiempos de la ejecución de la solución de cerrojo en zona fronteriza en Java, API Estándar.

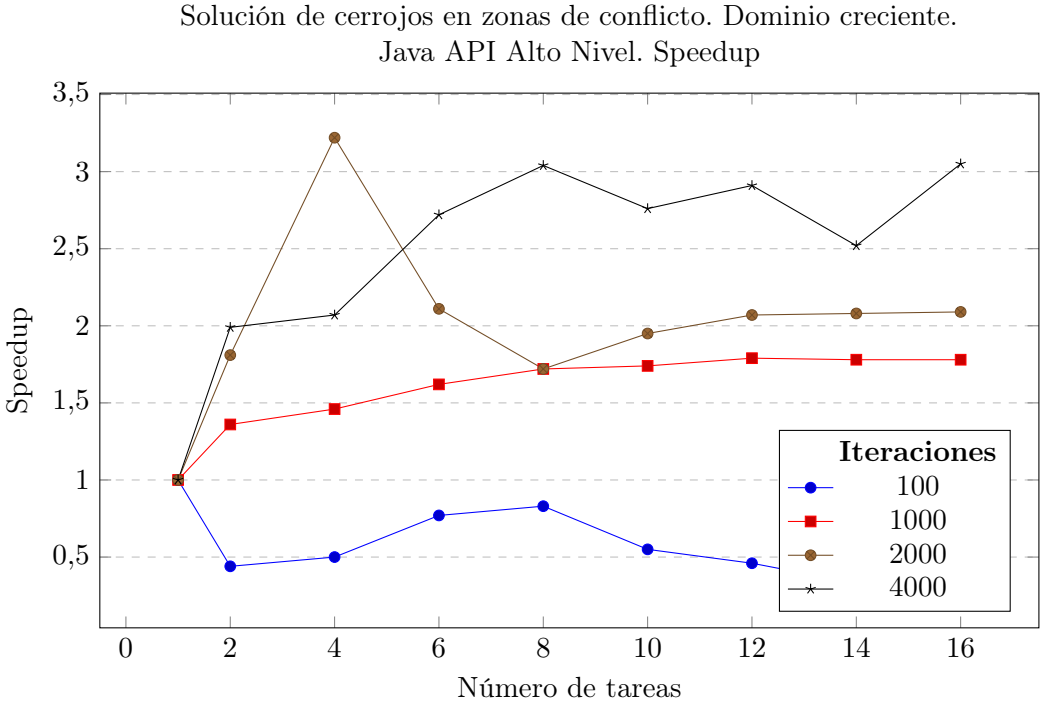

Figura B.13: Speedup de la ejecución de la solución de un cerrojo por frontera con dominio creciente en Java, API Alto Nivel.

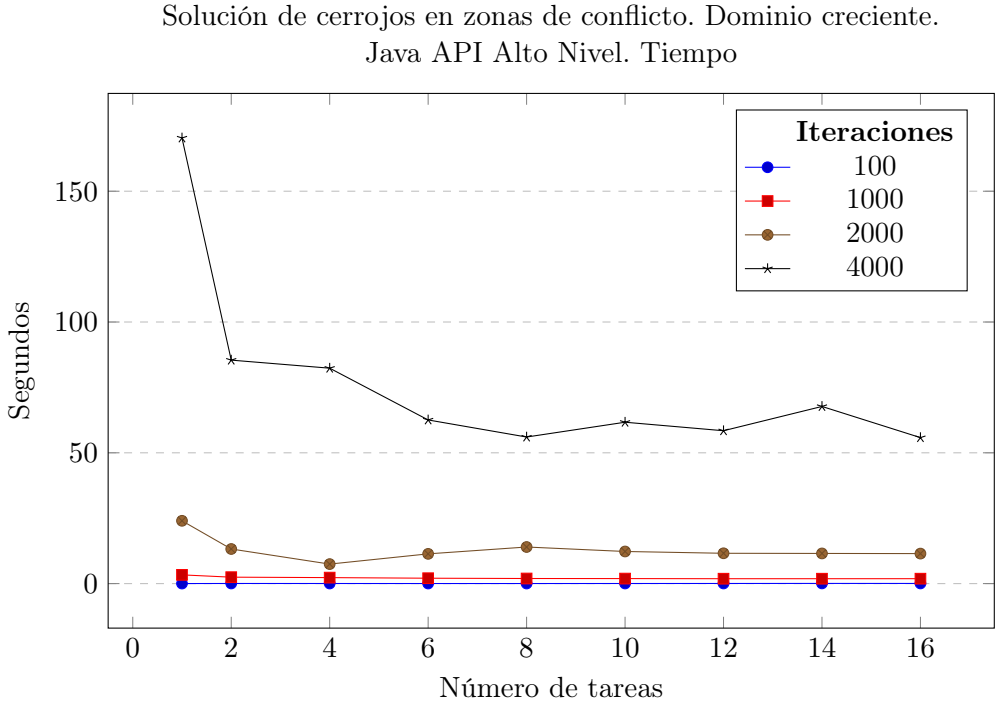

Figura B.14: Tiempos de la ejecución de la solución de un cerrojo por frontera con dominio creciente en Java, API Alto Nivel.

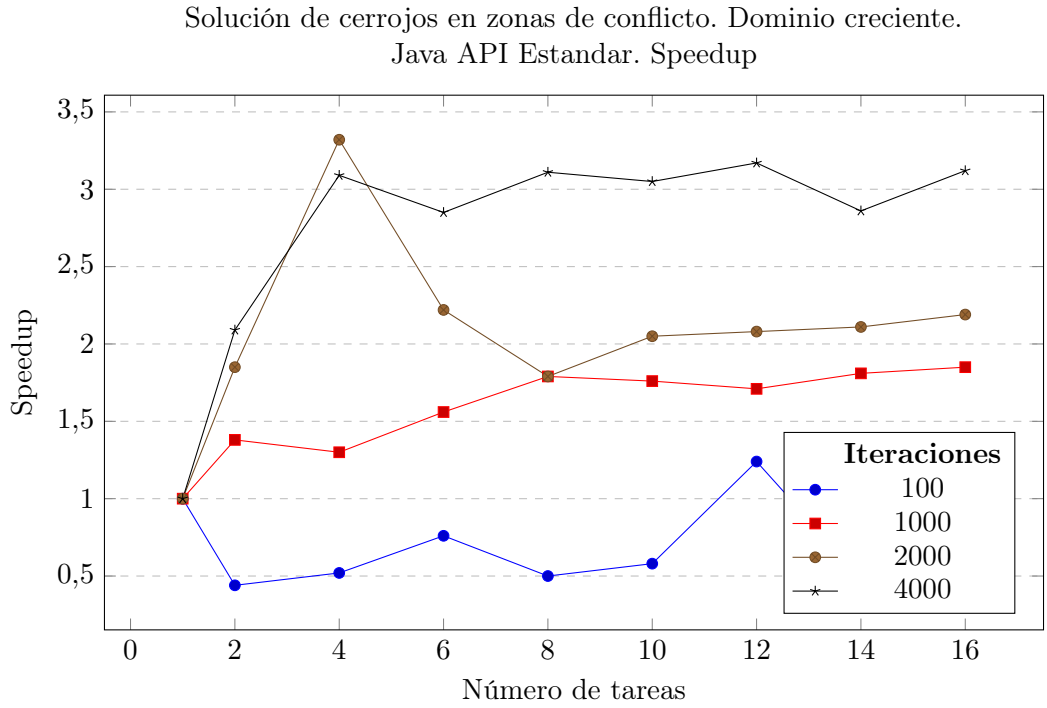

Figura B.15: Speedup de la ejecución de la solución de un cerrojo por frontera con dominio creciente en Java, API Estándar.

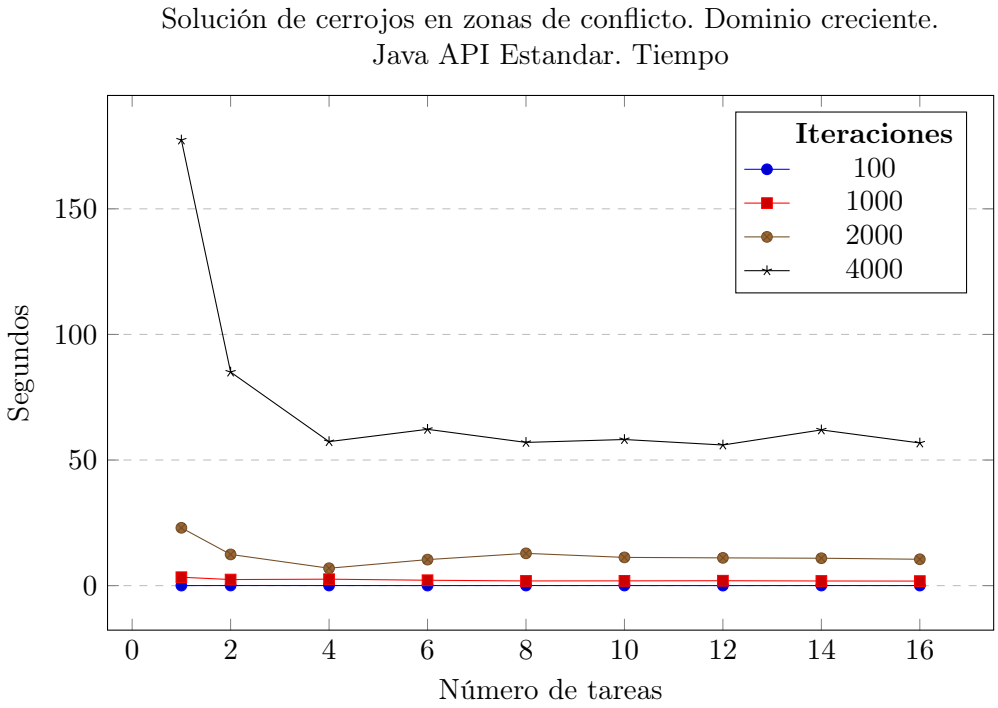

Figura B.16: Tiempos de la ejecución de la solución de un cerrojo por frontera con dominio creciente en Java, API Estándar.

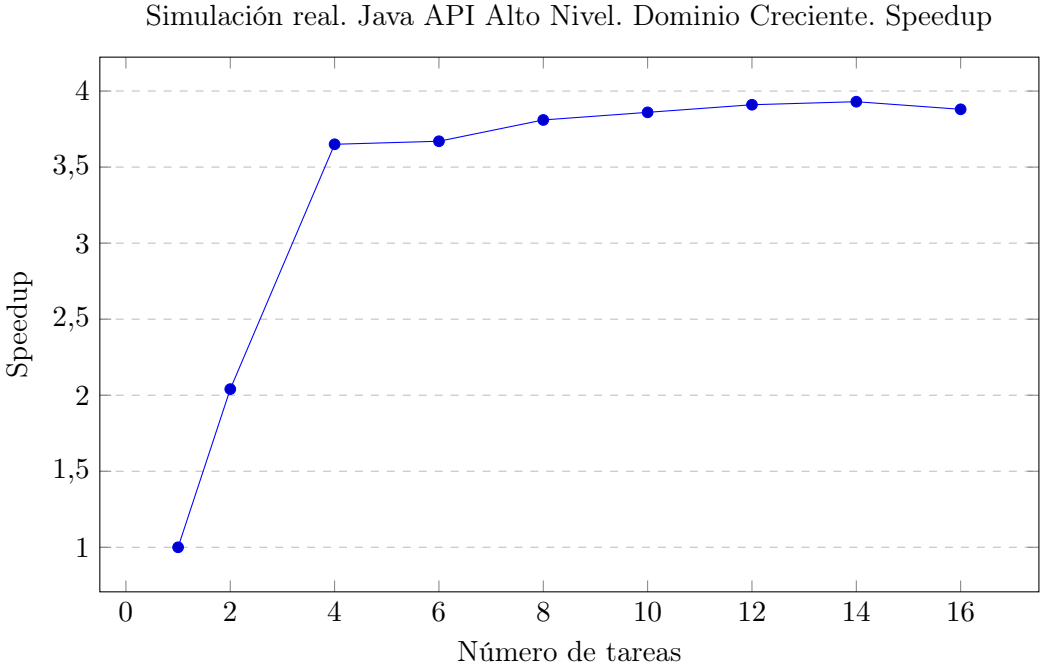

Figura B.17: Speedup de una simulación de 2 años de crecimiento tumoral, en la versión de dominio creciente, implementada en Java.

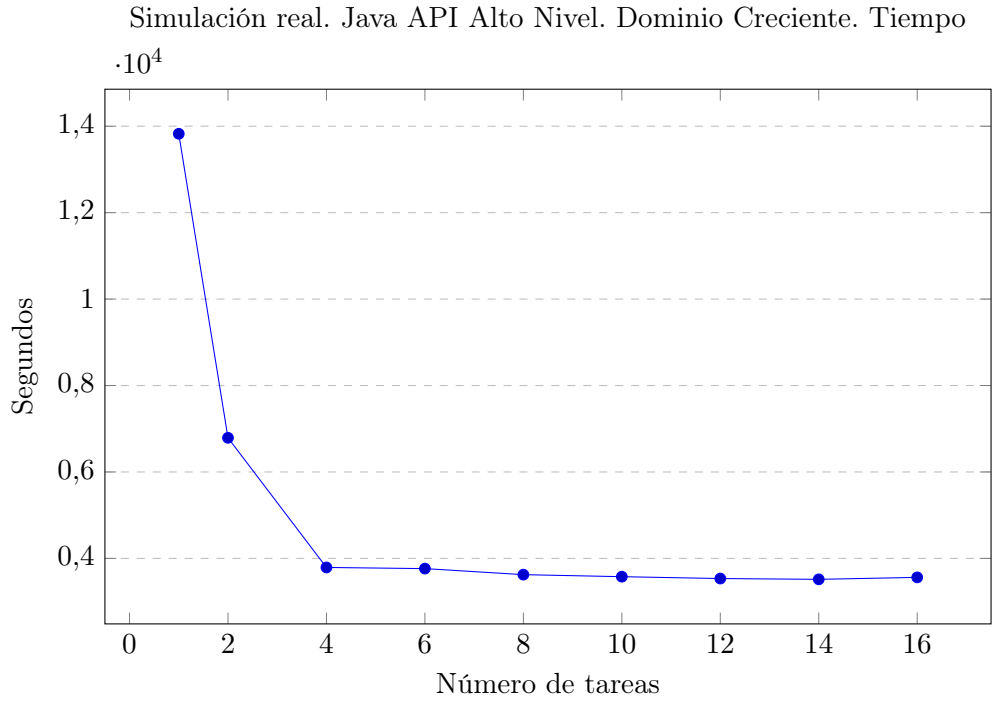

Figura B.18: Tiempos de una simulación de 2 años de crecimiento tumoral, en la versión de dominio creciente, implementada en Java.

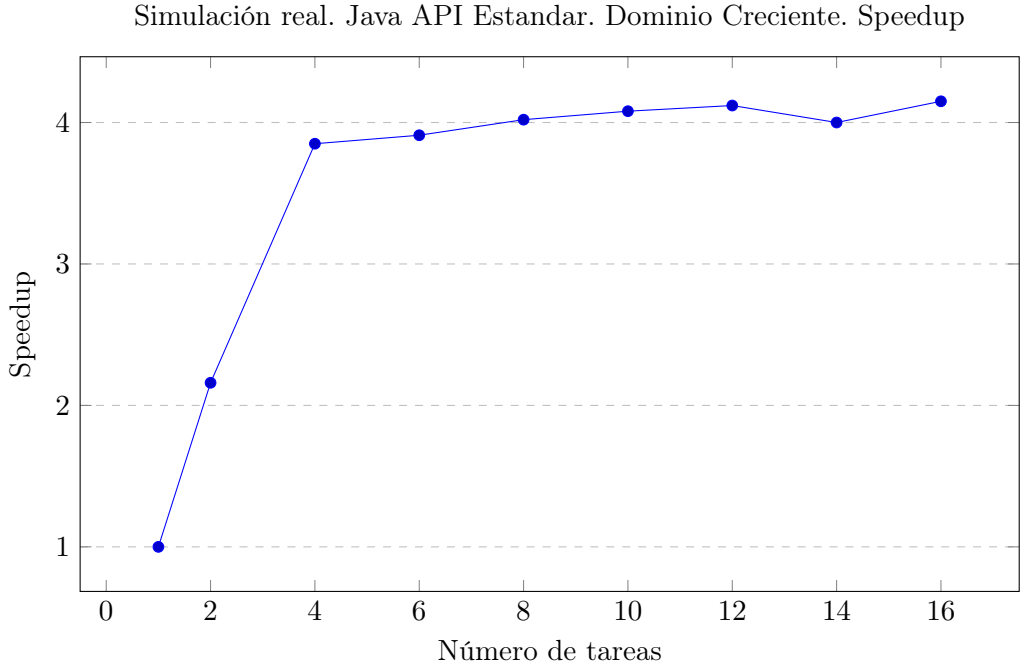

Figura B.19: Speedup de una simulación de 2 años de crecimiento tumoral, en la versión de dominio creciente, implementada en Java.

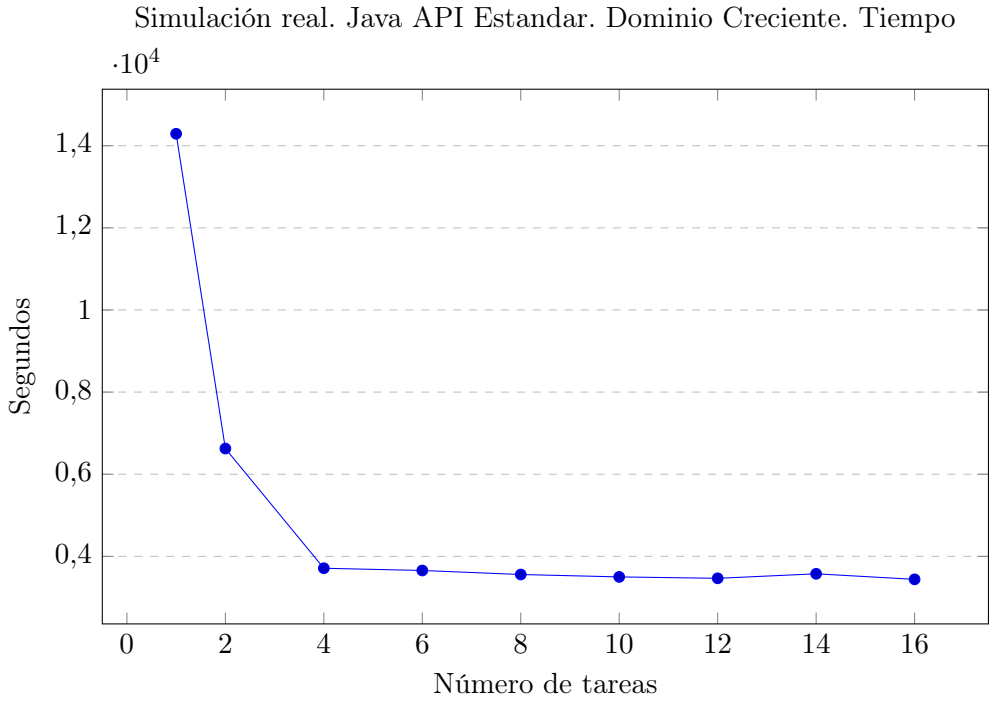

Figura B.20: Tiempos de una simulación de 2 años de crecimiento tumoral, en la versión de dominio creciente, implementada en Java.

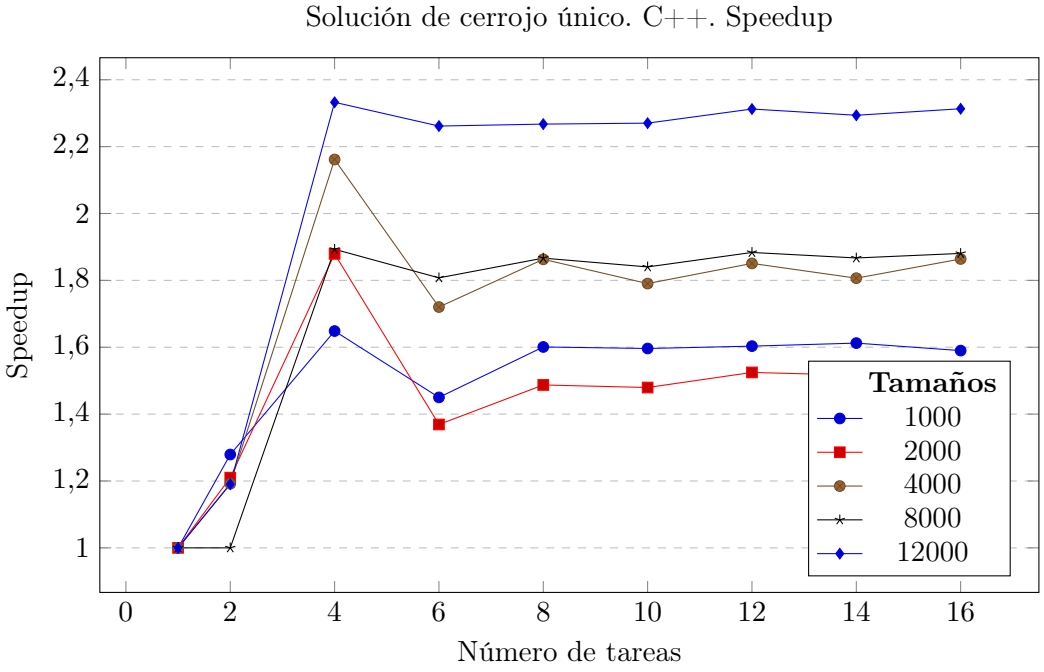

Figura B.21: Speedup de la ejecución de la solución de cerrojo único en C++.

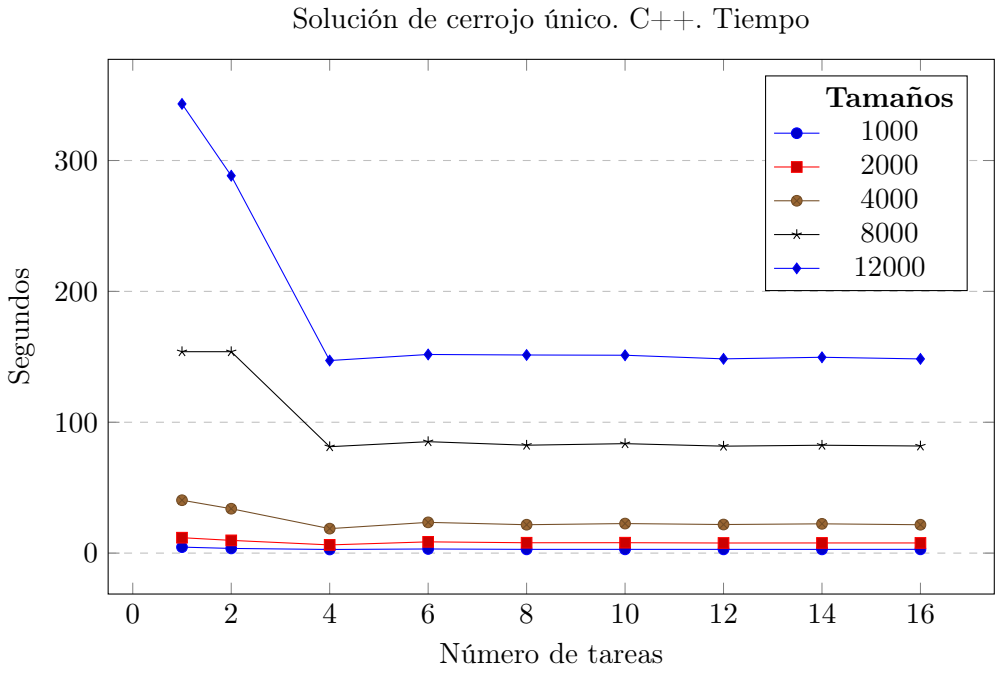

Figura B.22: Tiempos de la ejecución de la solución de cerrojo único en C++.

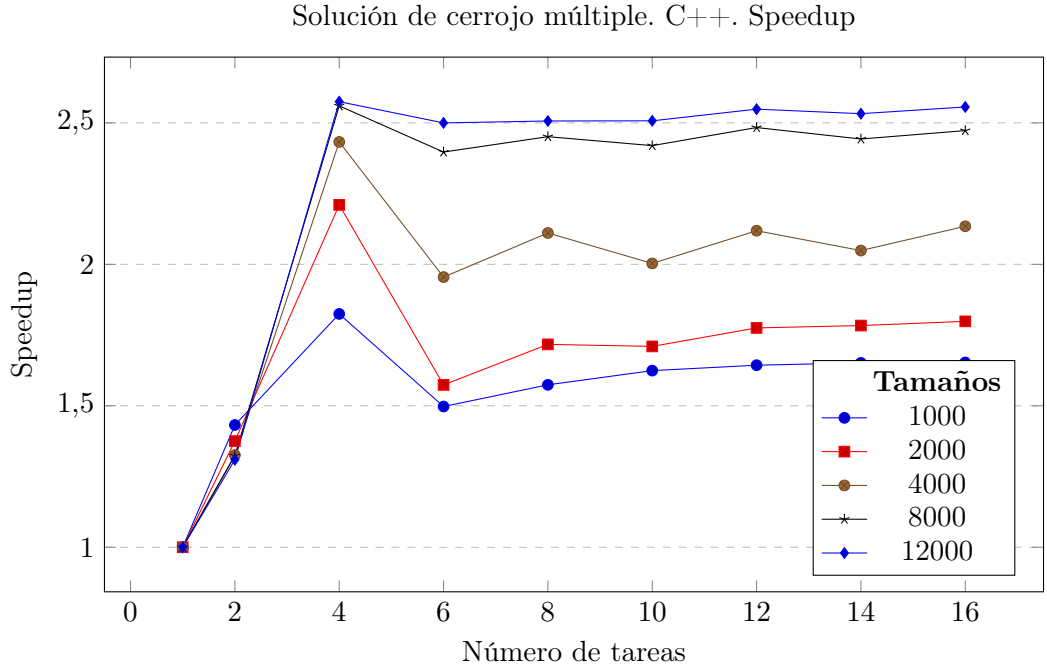

Figura B.23: Speedup de la ejecución de la solución de cerrojo múltiple en C++.

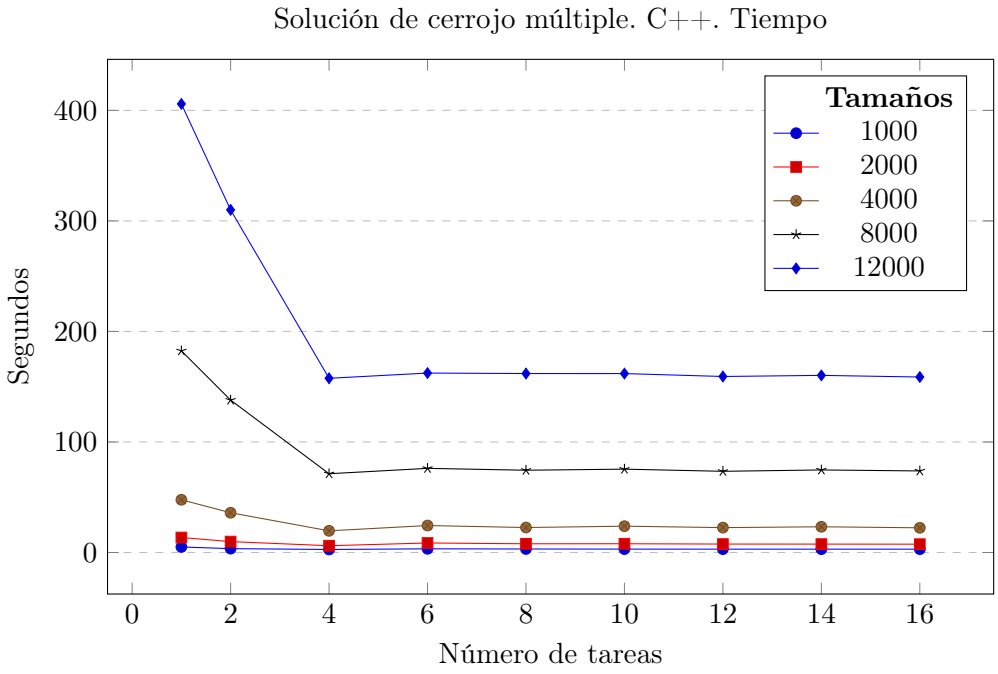

Figura B.24: Tiempos de la ejecución de la solución de cerrojo múltiple en C++.

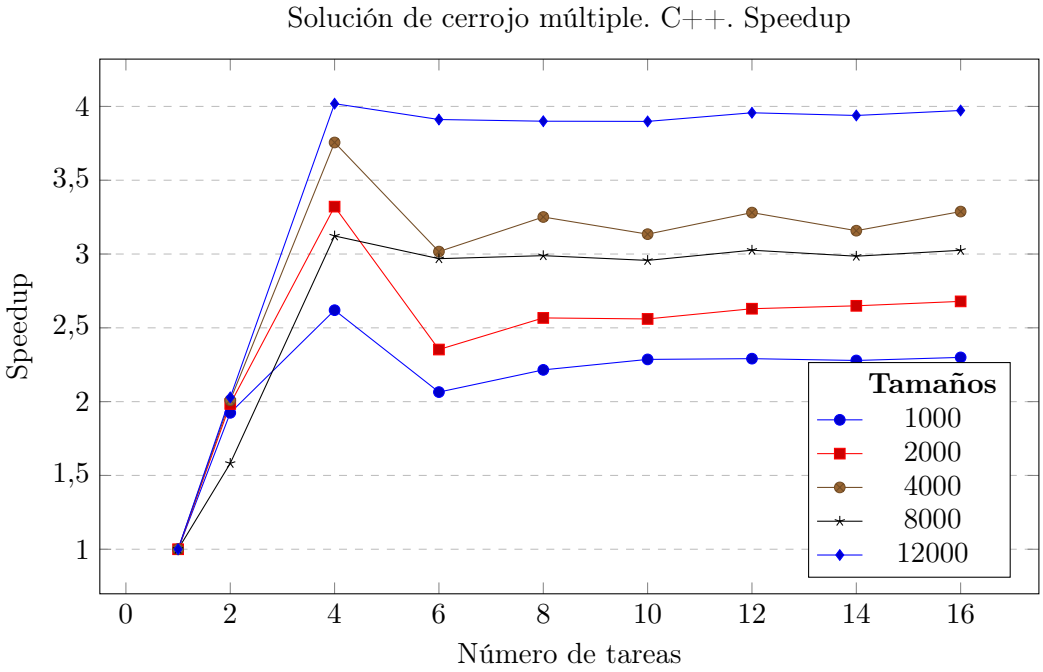

Figura B.25: Speedup de la ejecución de la solución de cerrojo en fronteras en C++.

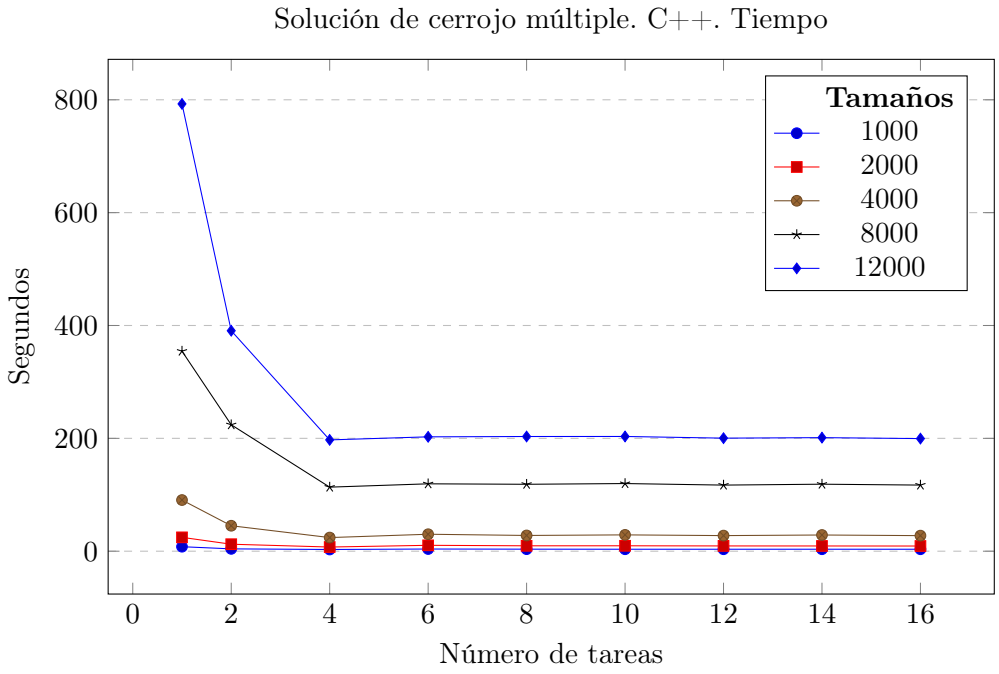

Figura B.26: Tiempos de la ejecución de la solución de cerrojo en fronteras en C++.

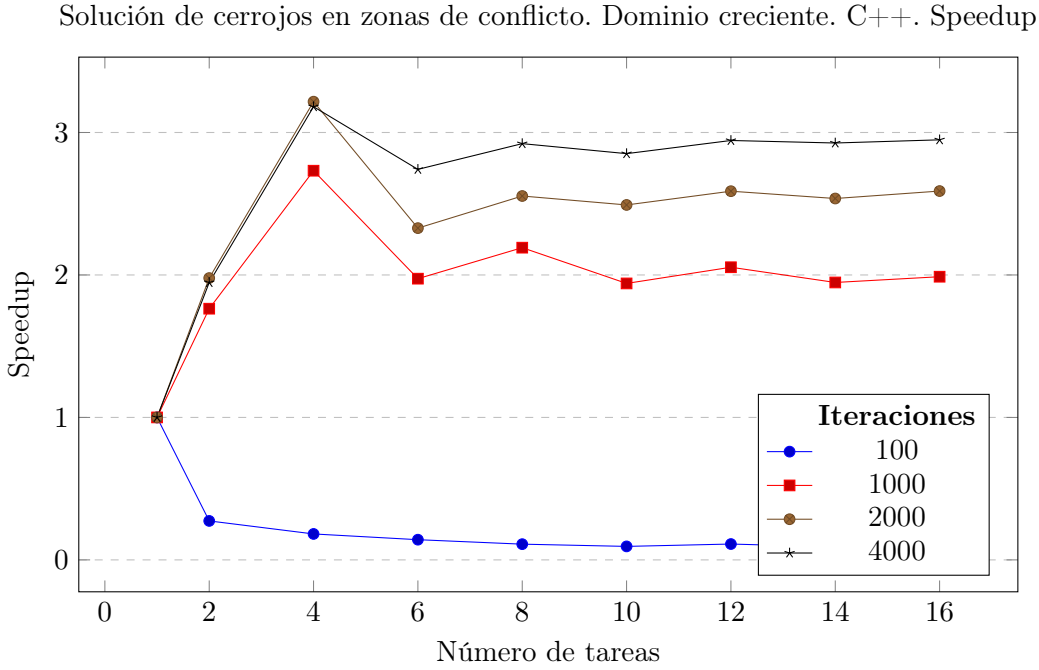

Figura B.27: Speedup de la ejecución de la solución de cerrojo en fronteras, con dominio de cómputo creciente, en C++.

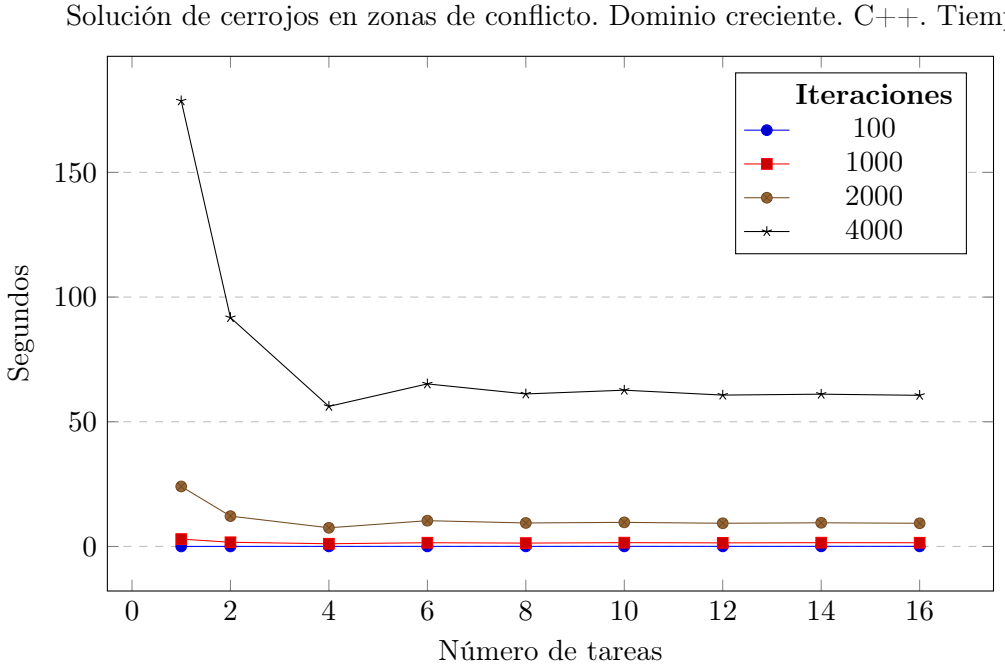

Figura B.28: Tiempos de la ejecución de la solución de cerrojo en fronteras, con dominio de cómputo creciente, en C++.

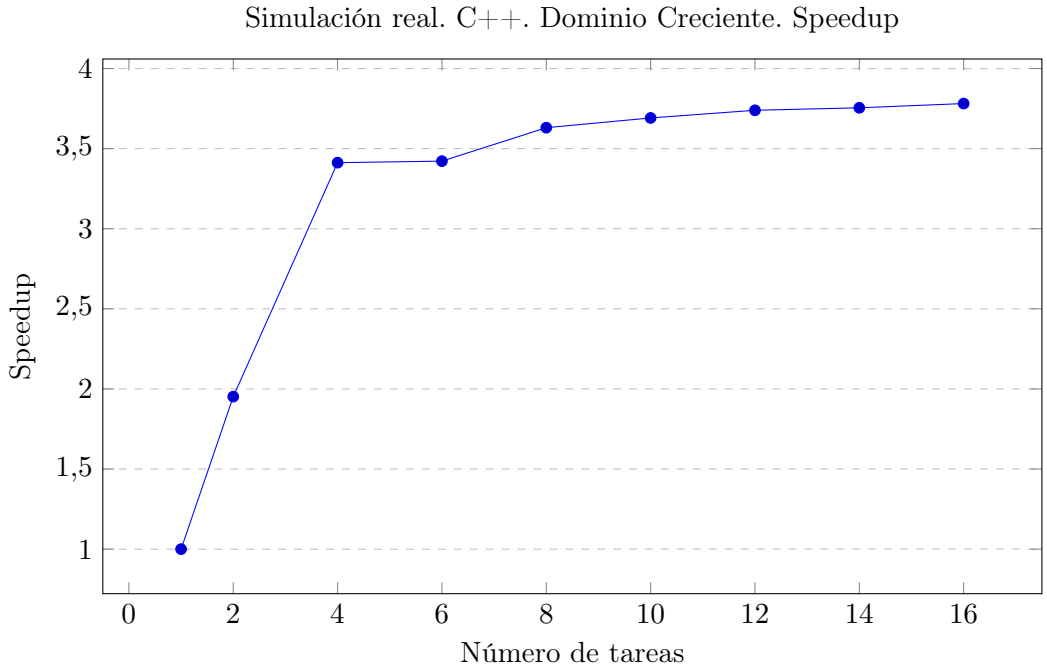

Figura B.29: Speedup de una simulación de 2 años de crecimiento tumoral, en la versión de dominio creciente, implementada en C++.

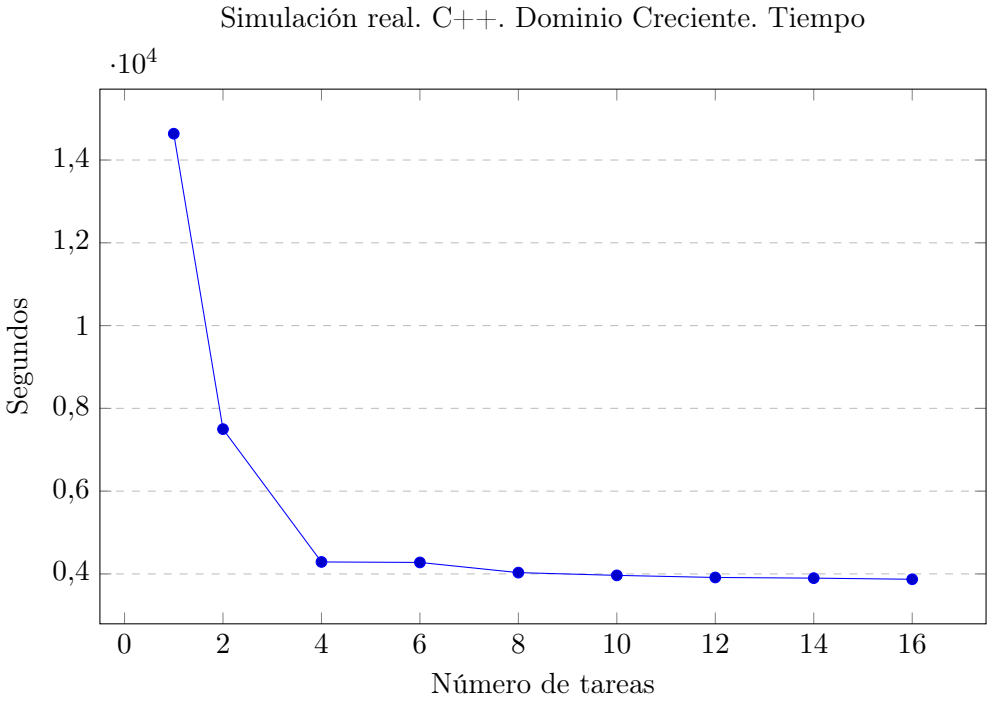

Figura B.30: Tiempos de una simulación de 2 años de crecimiento tumoral, en la versión de dominio creciente, implementada en C++.

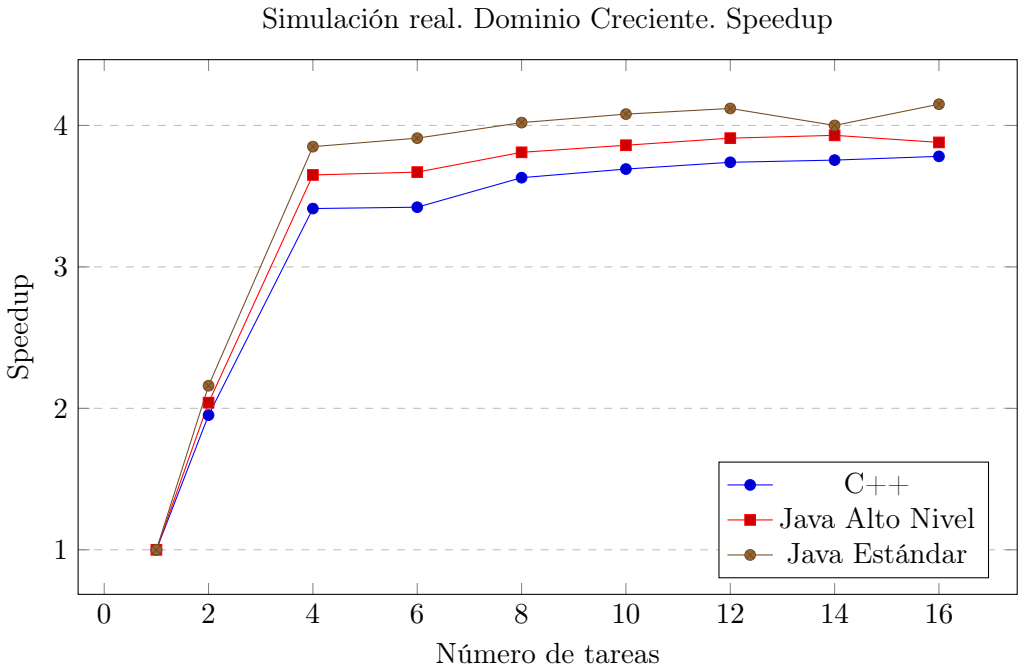

Figura B.31: Speedup de una simulación de 2 años de crecimiento tumoral, en la versión de dominio creciente. Comparativa entre C++ y Java. Pueden observarse mediciones muy similares.

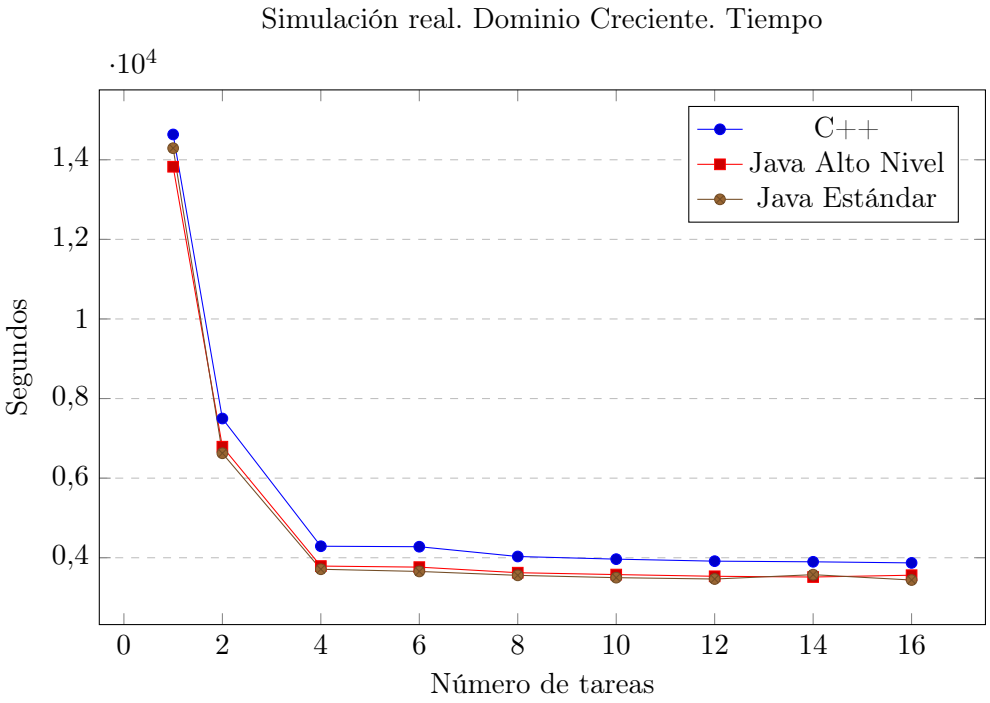

Figura B.32: Tiempos de una simulación de 2 años de crecimiento tumoral, en la versión de dominio creciente. Comparativa entre C++ y Java. Pueden observarse mediciones muy similares.

Apéndice C

Medidas en Clúster de Supercomputación

Se incluyen a continuación las gráficas de tiempo y *speedup* (no se usa el mejor algoritmo como numerador, sino la versión de una única tarea de cada uno). La carga paramétrica de todas las simulaciones se corresponde con $P_S = 1$, $P_P = 0{,}8$, $P_M = 0{,}2$, $NP = 5$ y $\rho = 2$. En las simulaciones de domino computacional estático se varía el tamaño del dominio tisular y se mantienen 1000 generaciones. En las simulaciones de domino computacional dinámico se mantiene un tamaño de 8000^2 células y se varía el número de iteraciones. Por limitaciones técnicas ajenas al autor, únicamente se han usado las versiones implementadas en Java.

Las pruebas presentadas se realizan en Java, versión 1.7.0_40 de la OpenJDK, sobre Redhat Enterprise 6.4 bajo el kernel 2.6.32-358 (64 bits) de Linux. Los nodos disponen de dos procesadores Intel® Xeon™ E5-2670 @2.6GHz, *octa-core*, *hyperth-reading* desactivado, con un total de 16 *cores* por cada nodo, 2MB de caché L2 y 20MB de caché L3 (por procesador), junto a 128GB de memoria RAM (por nodo).

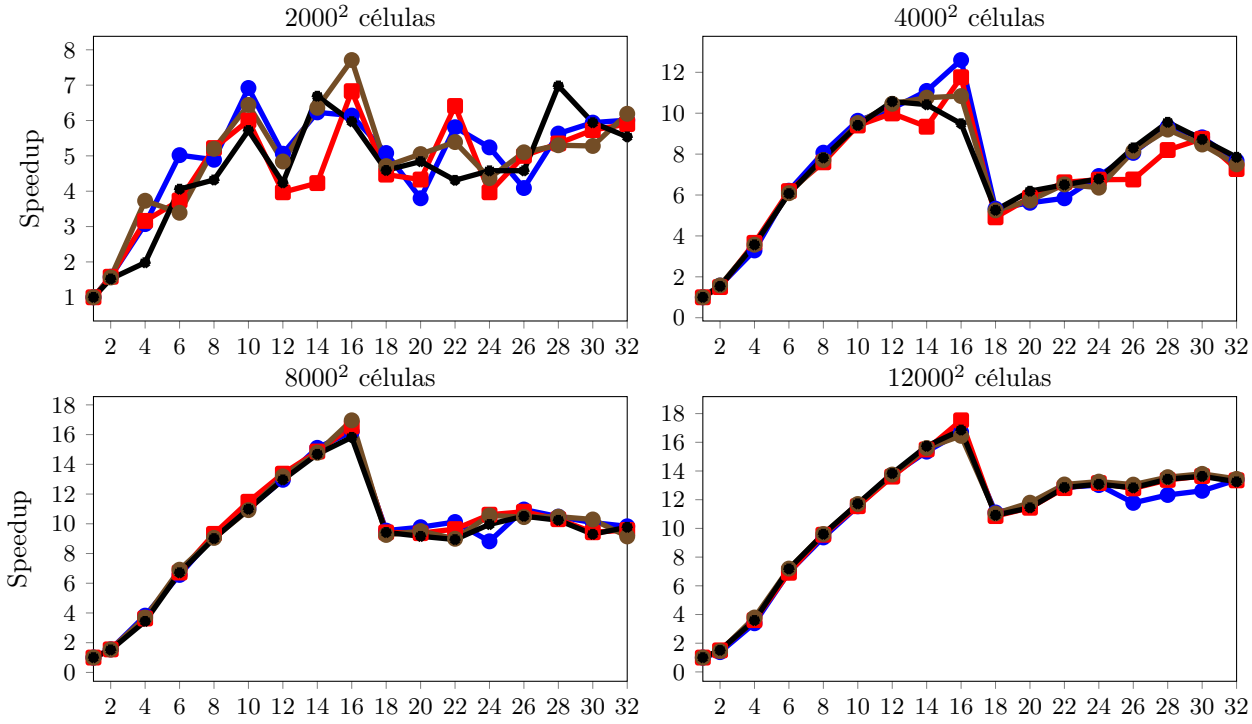

Figura C.1: *Speedups* obtenidos para la versión de cerrojo único en Java, API Estándar, para diferentes tamaños de dominio tisular. Ejecutado en 4 nodos del clúster de supercomputación de la UCA. Se presenta una curva por cada nodo.

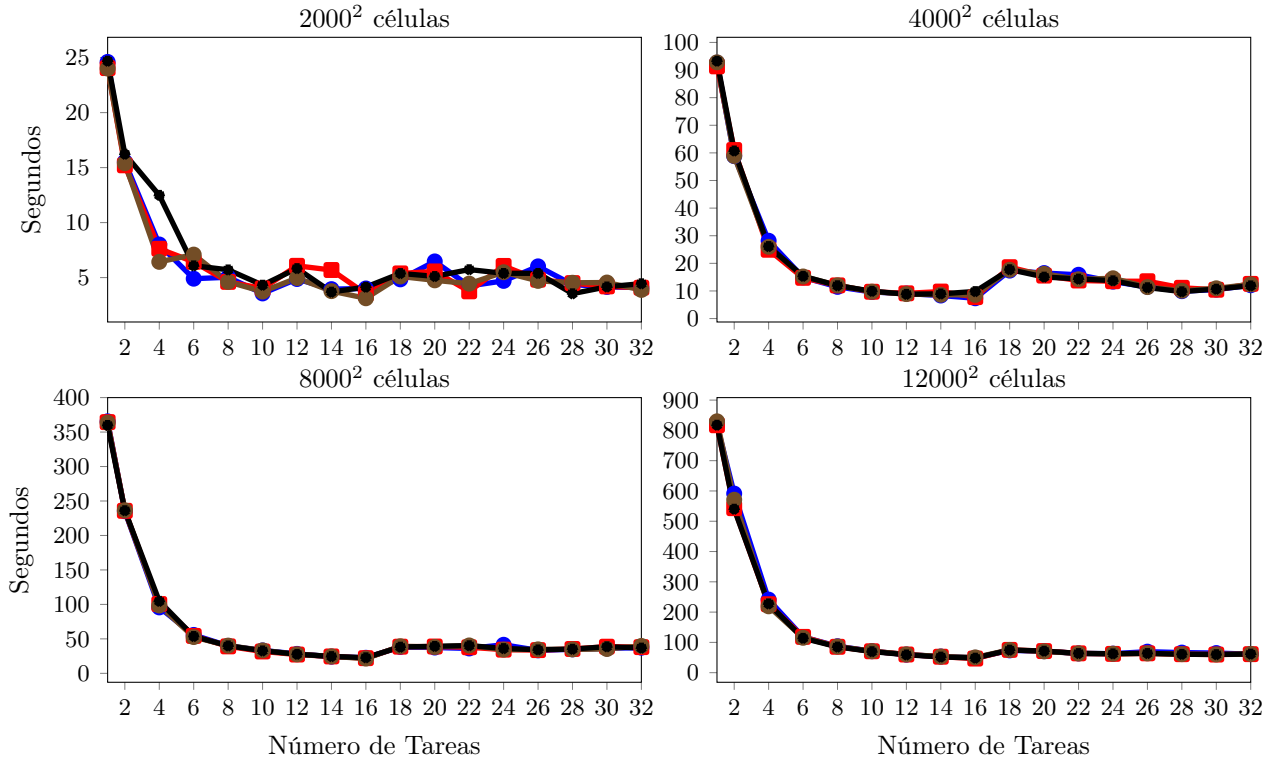

Figura C.2: Tiempos obtenidos para la versión de cerrojo único en Java, API Estándar, para diferentes tamaños de dominio tisular. Ejecutado en 4 nodos del clúster de supercomputación de la UCA. Se presenta una curva por cada nodo.

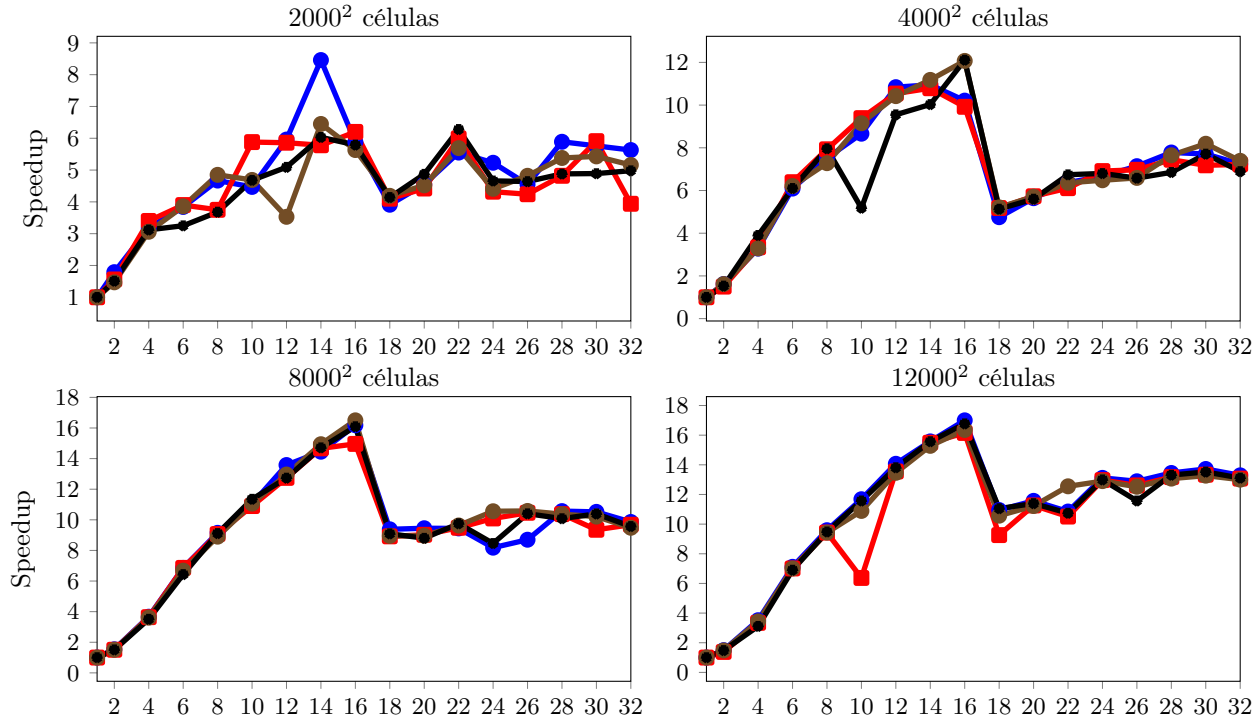

Figura C.3: *Speedups* obtenidos para la versión de cerrojo único en Java, API de Alto Nivel, para diferentes tamaños de dominio tisular. Ejecutado en 4 nodos del clúster de supercomputación de la UCA. Se presenta una curva por cada nodo.

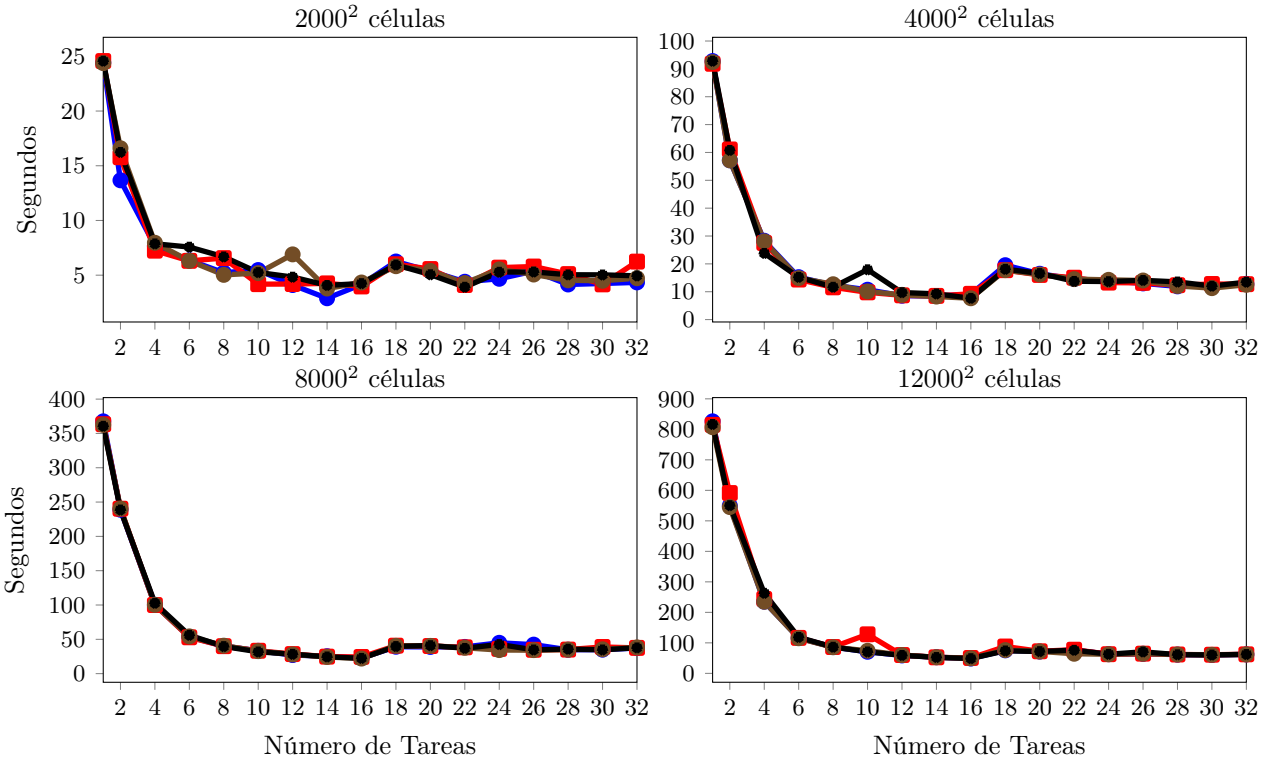

Figura C.4: Tiempos obtenidos para la versión de cerrojo único en Java, API de Alto Nivel, para diferentes tamaños de dominio tisular. Ejecutado en 4 nodos del clúster de supercomputación de la UCA. Se presenta una curva por cada nodo.

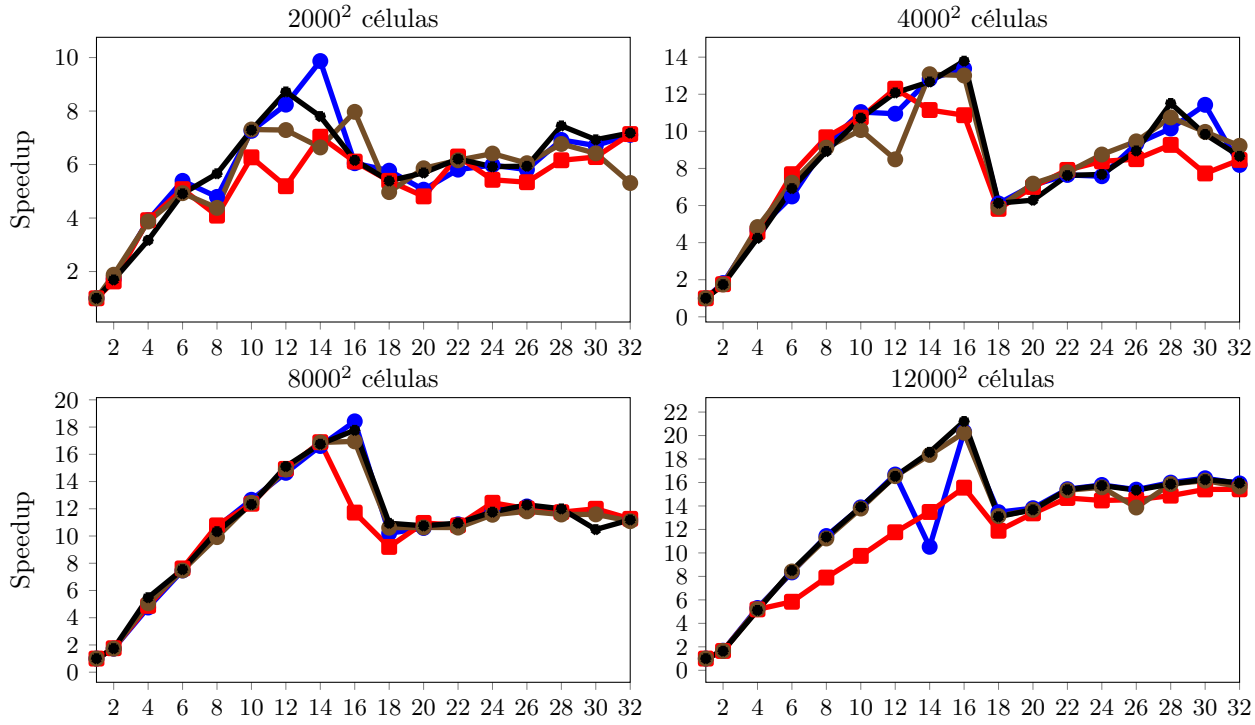

Figura C.5: *Speedups* obtenidos para la versión de un cerrojo por partición en Java, API Estándar, para diferentes tamaños de dominio tisular. Ejecutado en 4 nodos del clúster de supercomputación de la UCA. Se presenta una curva por cada nodo.

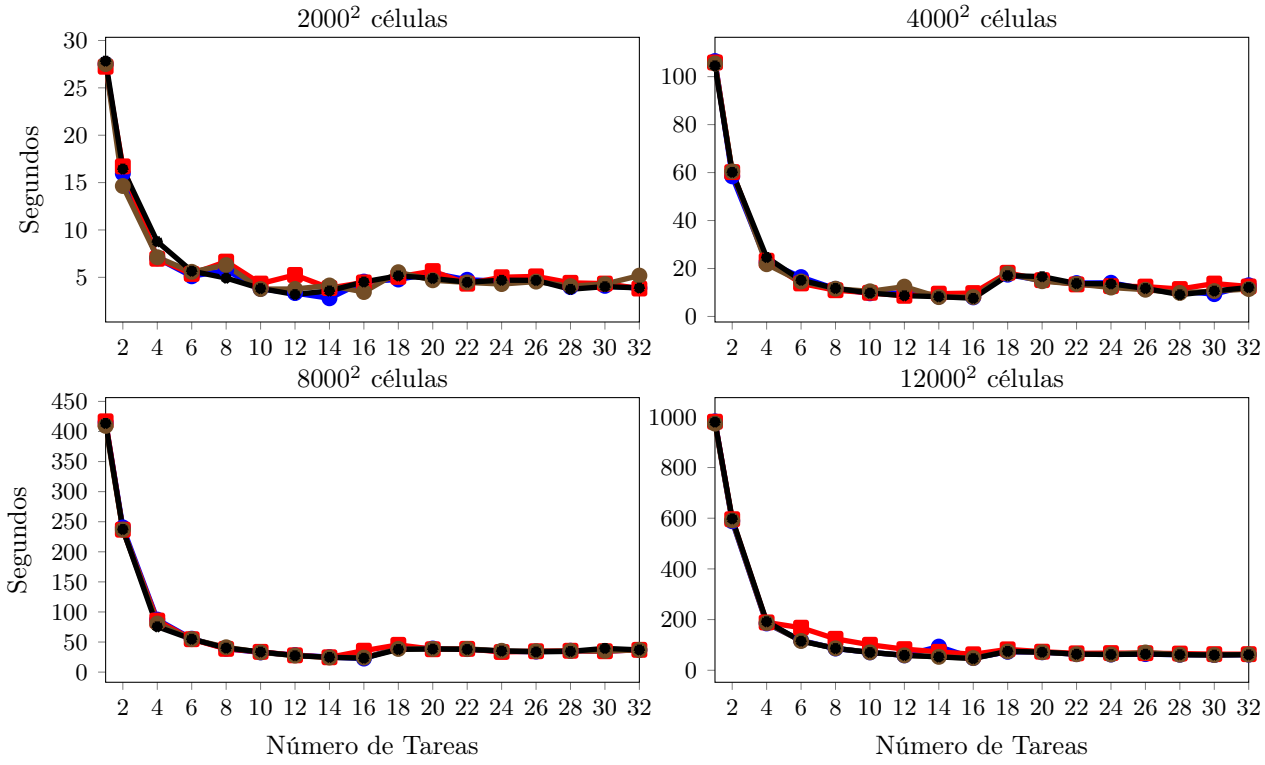

Figura C.6: Tiempos obtenidos para la versión de un cerrojo por partición en Java, API Estándar, para diferentes tamaños de dominio tisular. Ejecutado en 4 nodos del clúster de supercomputación de la UCA. Se presenta una curva por cada nodo.

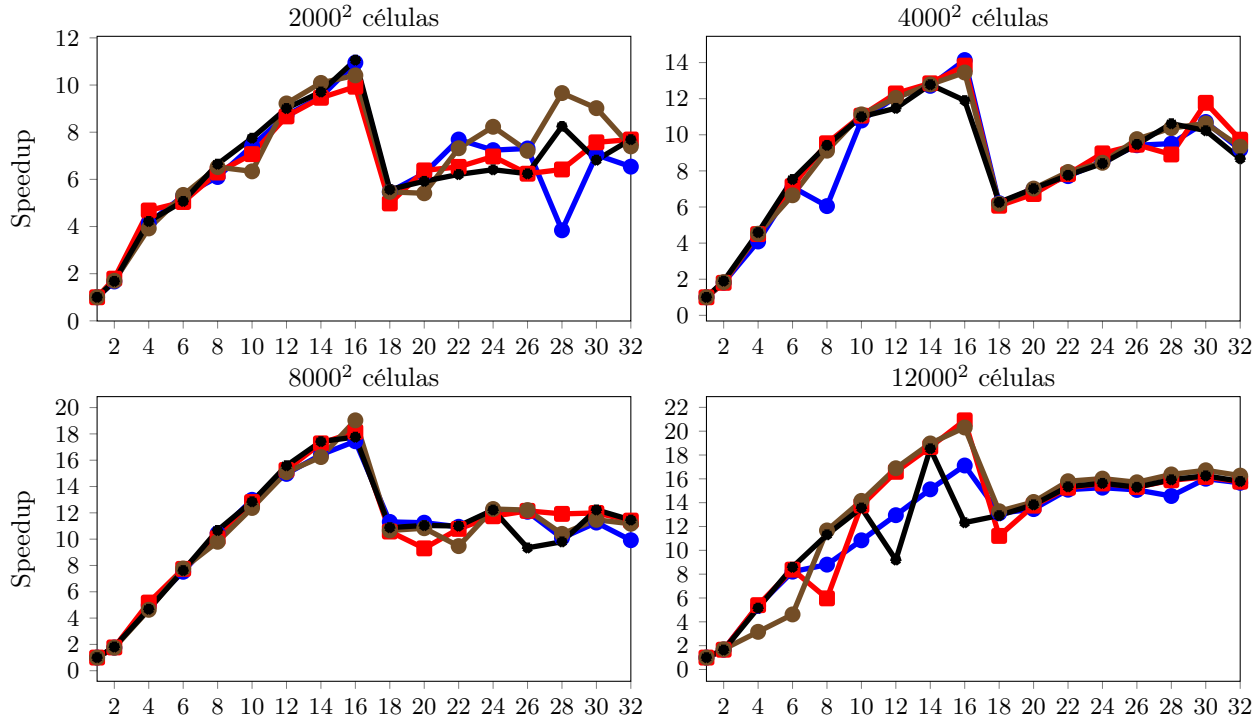

Figura C.7: *Speedups* obtenidos para la versión de un cerrojo por partición en Java, API de Alto Nivel, para diferentes tamaños de dominio tisular. Ejecutado en 4 nodos del clúster de supercomputación de la UCA. Se presenta una curva por cada nodo.

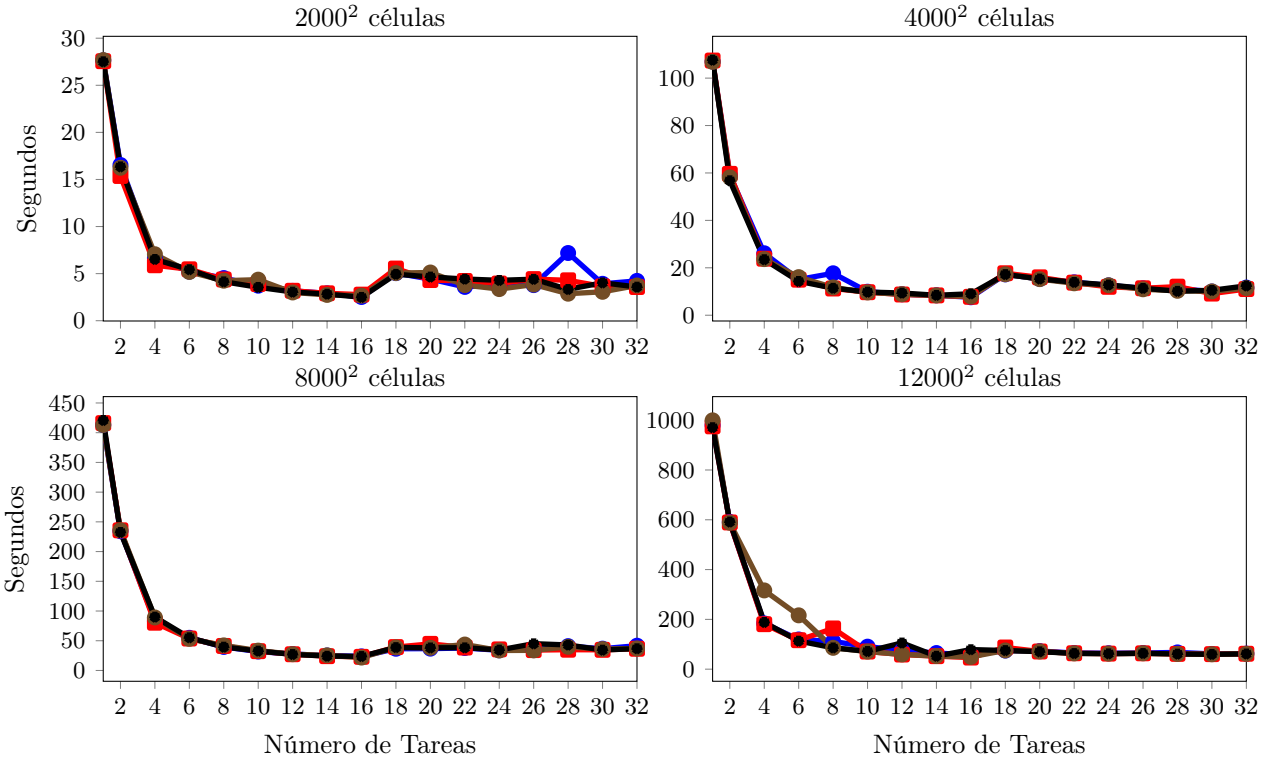

Figura C.8: Tiempos obtenidos para la versión de un cerrojo por partición en Java, API de Alto Nivel, para diferentes tamaños de dominio tisular. Ejecutado en 4 nodos del clúster de supercomputación de la UCA. Se presenta una curva por cada nodo.

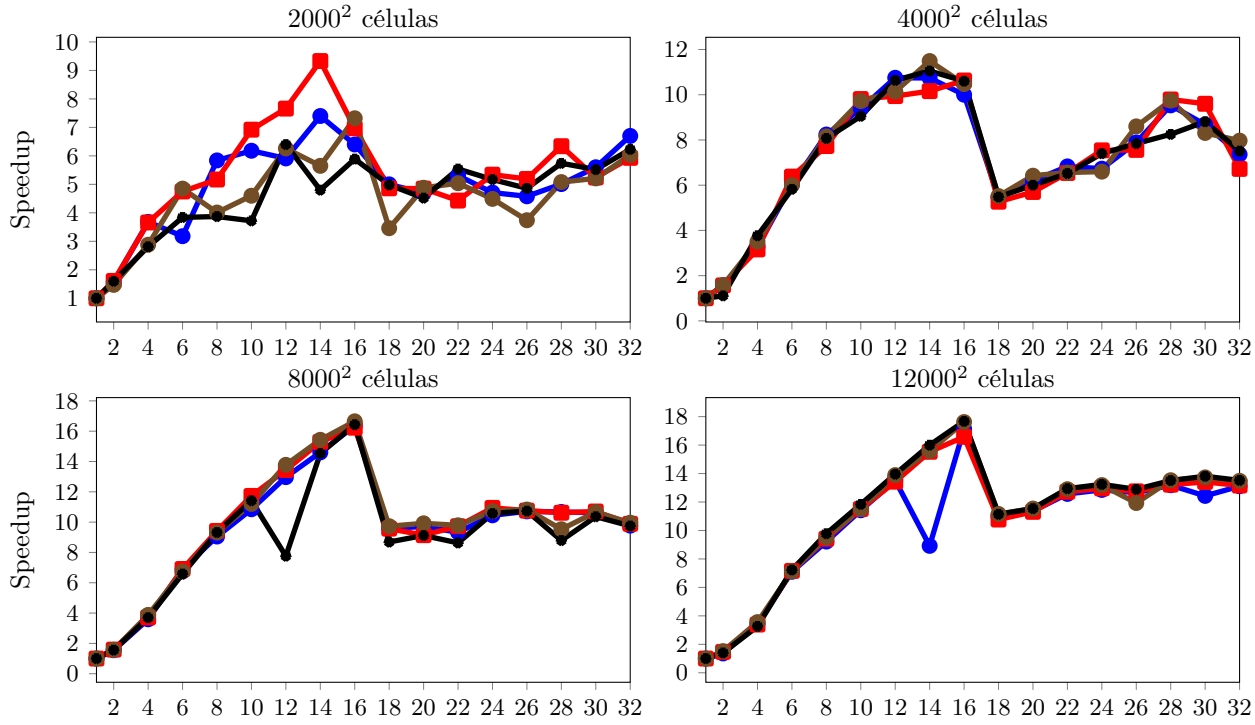

Figura C.9: *Speedups* obtenidos para la versión de cerrojos para las fronteras en Java, API Estándar, para diferentes tamaños de dominio tisular. Ejecutado en 4 nodos del clúster de supercomputación de la UCA. Se presenta una curva por cada nodo.

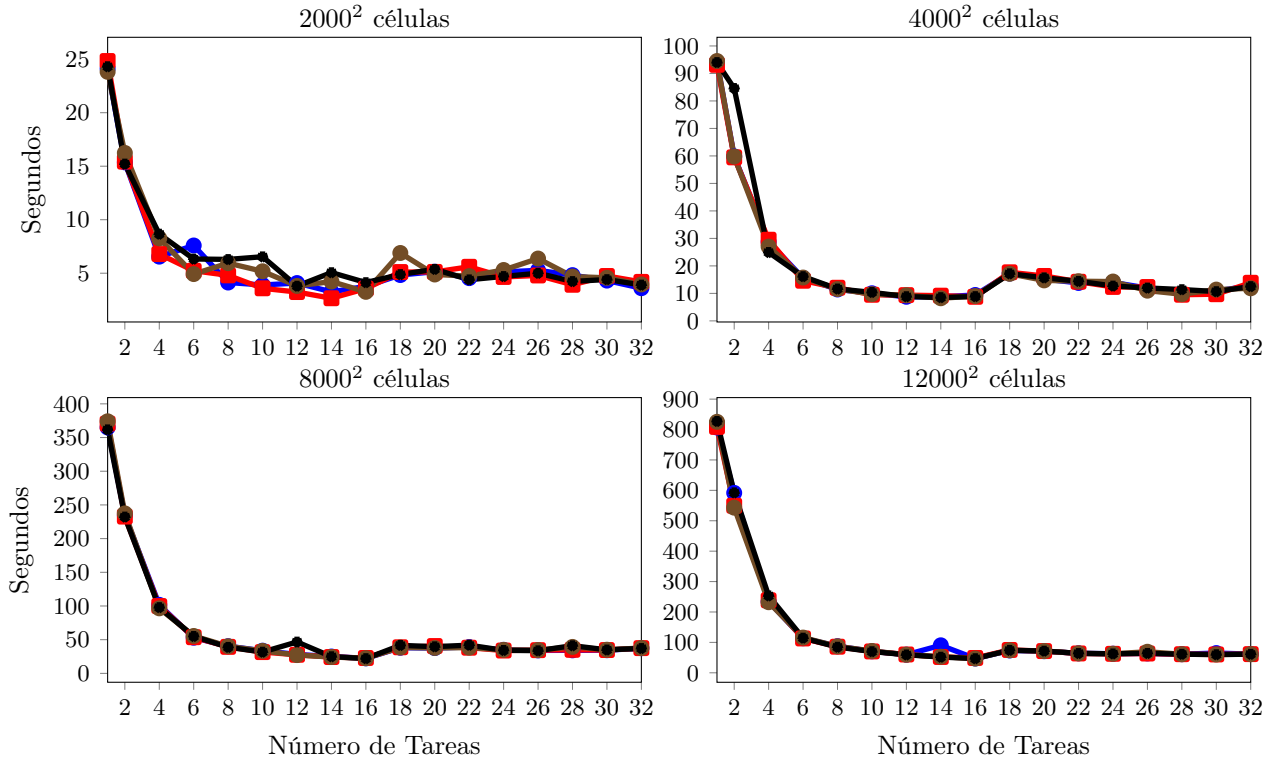

Figura C.10: Tiempos obtenidos para la versión de cerrojos para las fronteras en Java, API Estándar, para diferentes tamaños de dominio tisular. Ejecutado en 4 nodos del clúster de supercomputación de la UCA. Se presenta una curva por cada nodo.

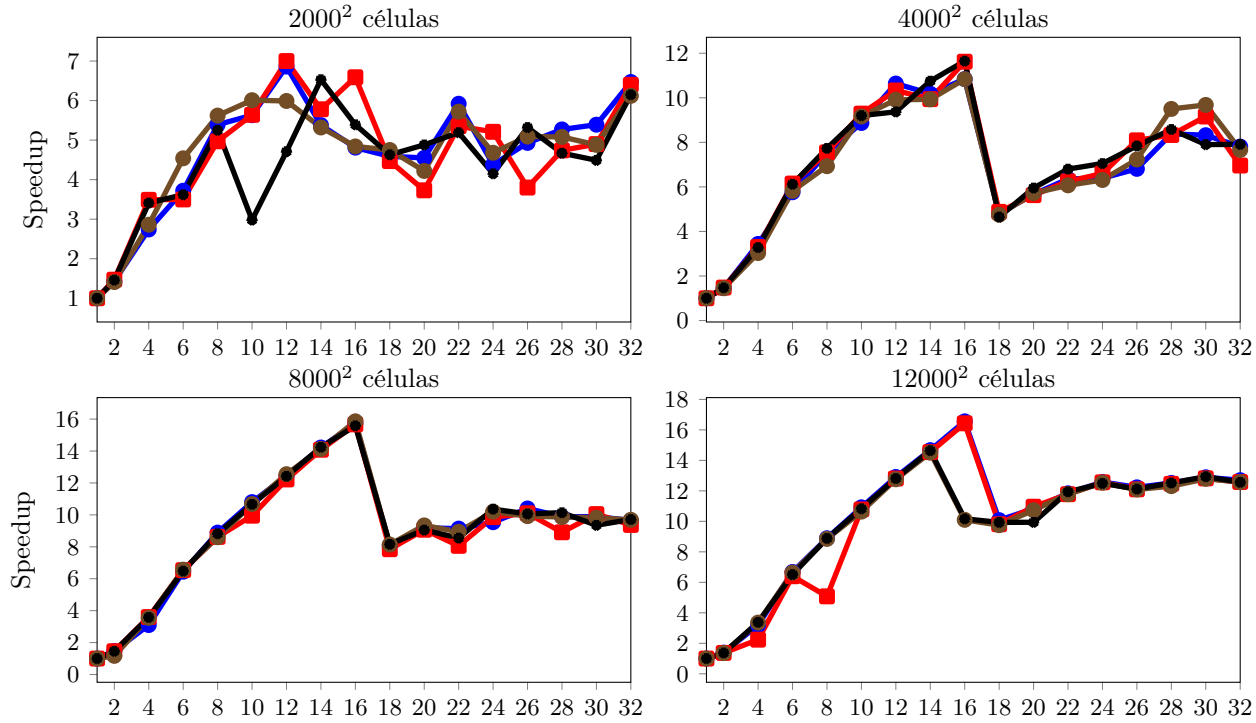

Figura C.11: *Speedups* obtenidos para la versión de cerrojos para las fronteras en Java, API de Alto Nivel, para diferentes tamaños de dominio tisular. Ejecutado en 4 nodos del clúster de supercomputación de la UCA. Se presenta una curva por cada nodo.

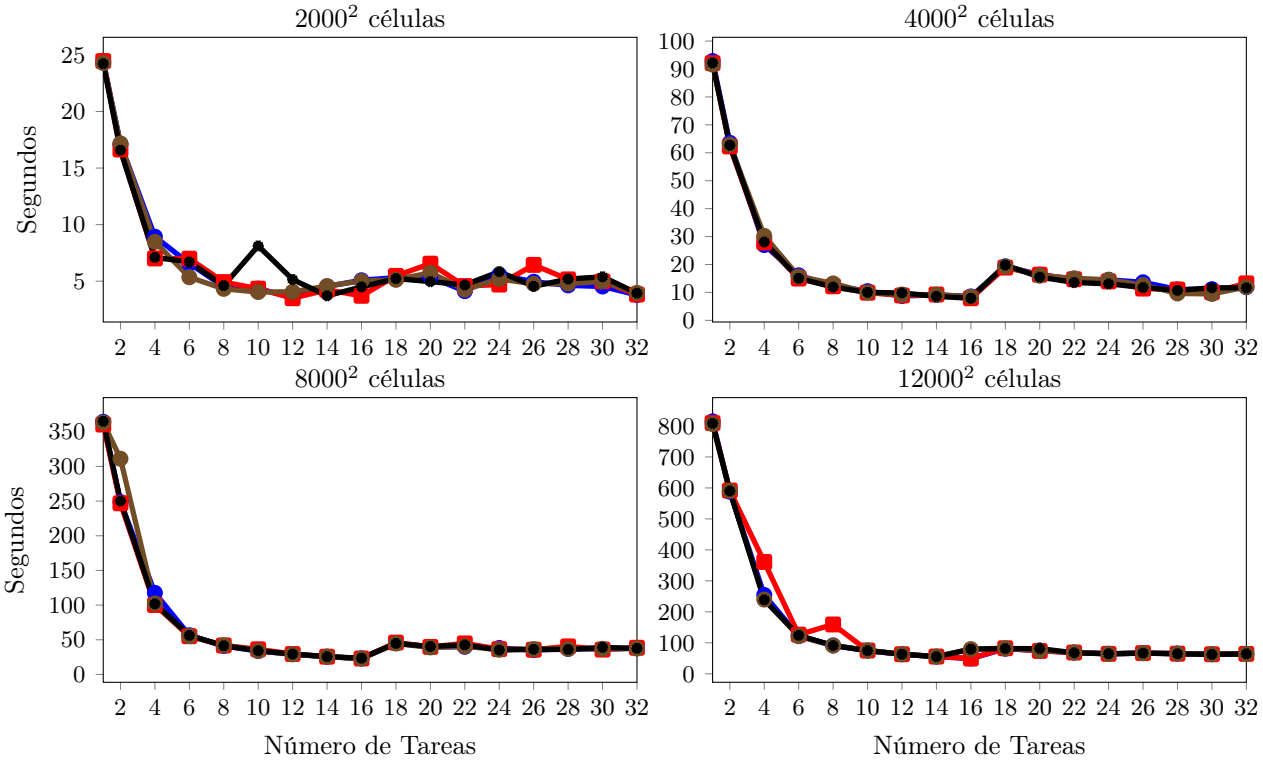

Figura C.12: Tiempos obtenidos para la versión de cerrojos para las fronteras en Java, API de Alto Nivel, para diferentes tamaños de dominio tisular. Ejecutado en 4 nodos del clúster de supercomputación de la UCA. Se presenta una curva por cada nodo.

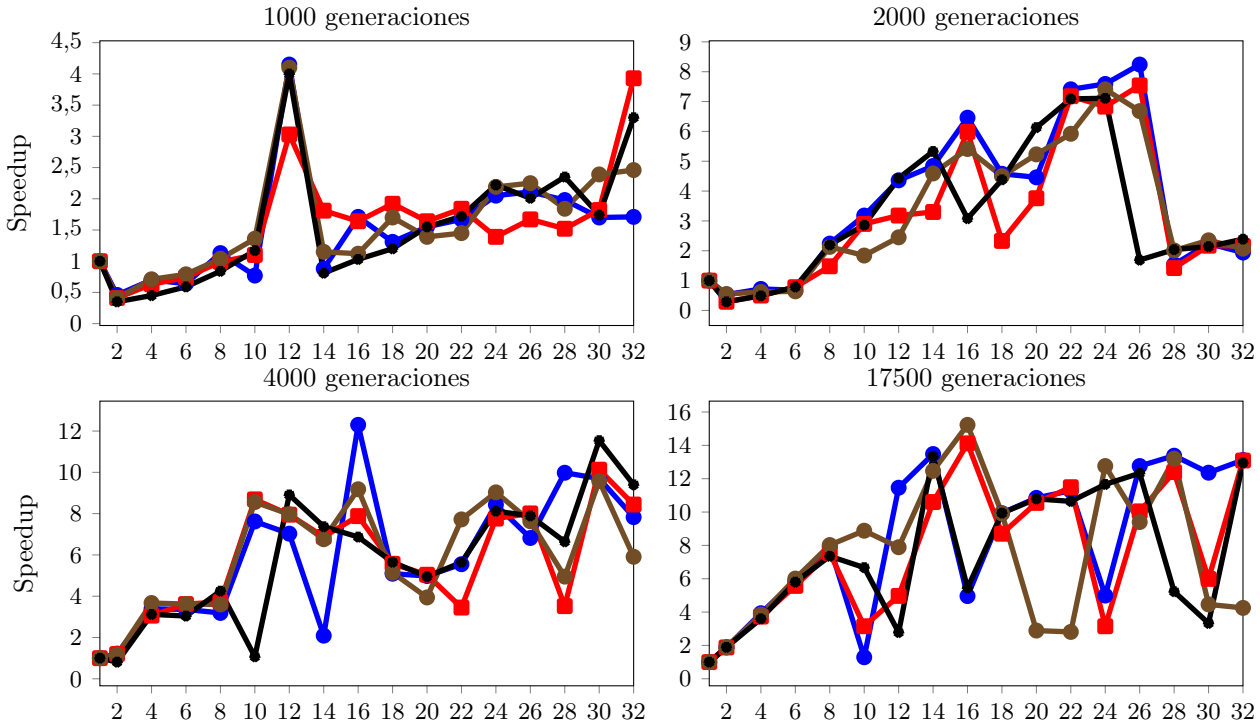

Figura C.13: *Speedups* obtenidos para la versión de cerrojos para las fronteras en Java, API Estándar, con dominio de cómputo creciente, para diferente número de pasos de tiempo discreto. Ejecutado en 4 nodos del clúster de supercomputación de la UCA. Se presenta una curva por cada nodo.

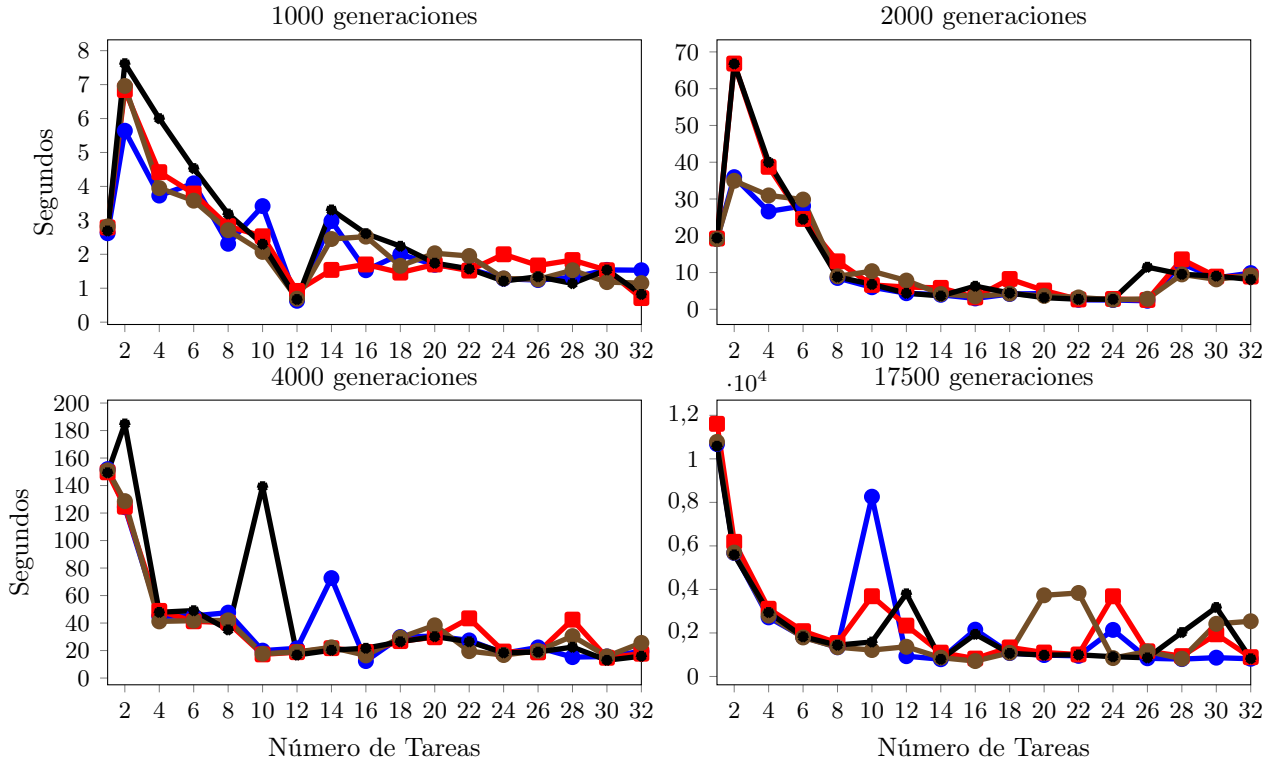

Figura C.14: Tiempos obtenidos para la versión de cerrojos para las fronteras en Java, API Estándar, con dominio de cómputo creciente, para diferente número de pasos de tiempo discreto. Ejecutado en 4 nodos del clúster de supercomputación de la UCA. Se presenta una curva por cada nodo.

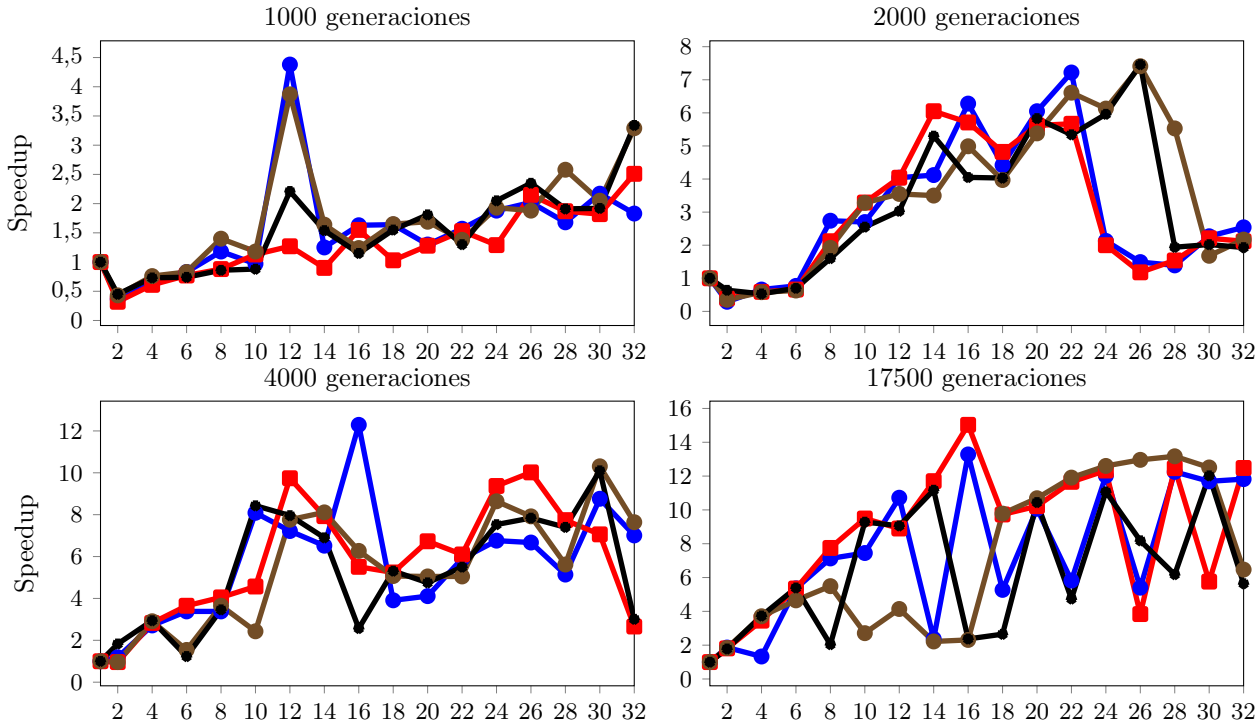

Figura C.15: *Speedups* obtenidos para la versión de cerrojos para las fronteras en Java, API de Alto Nivel, con dominio de cómputo creciente, para diferente número de pasos de tiempo discreto. Ejecutado en 4 nodos del clúster de supercomputación de la UCA. Se presenta una curva por cada nodo.

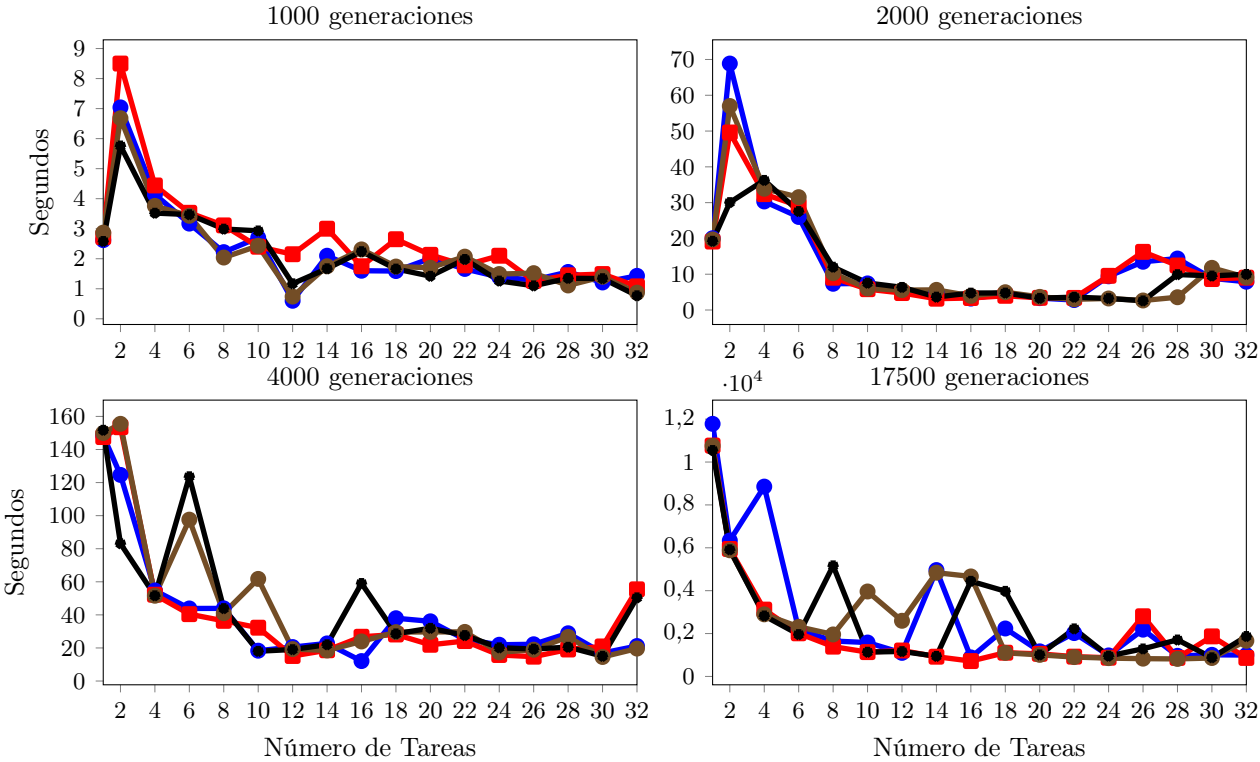

Figura C.16: Tiempos obtenidos para la versión de cerrojos para las fronteras en Java, API de Alto Nivel, con dominio de cómputo creciente, para diferente número de pasos de tiempo discreto. Ejecutado en 4 nodos del clúster de supercomputación de la UCA. Se presenta una curva por cada nodo.

Apéndice D

Descripción de Medidas en Clúster de Supercomputación

Se incluyen en esta sección las medias y desviaciones típicas calculadas a partir de las ejecuciones en distintos nodos del clúster. Cada conjunto de tablas se corresponde con las figuras de *speedup* y tiempo del anexo C.

2000^2 células

	Speedup		Tiempo	
Tareas	μ	σ	μ	σ
1	1,00	0,00	24,34	0,36
2	1,56	0,03	15,59	0,44
4	2,99	0,73	8,64	2,65
6	4,05	0,70	6,13	0,92
8	4,91	0,42	4,99	0,52
10	6,27	0,52	3,91	0,33
12	4,53	0,51	5,43	0,61
14	5,88	1,12	4,28	0,95
16	6,66	0,79	3,70	0,47
18	4,71	0,26	5,17	0,26
20	4,50	0,56	5,47	0,74
22	5,48	0,89	4,54	0,84
24	4,54	0,53	5,41	0,56
26	4,69	0,46	5,23	0,61
28	5,82	0,79	4,24	0,47
30	5,72	0,31	4,27	0,19
32	5,91	0,27	4,13	0,24

4000^2 células

	Speedup		Tiempo	
Tareas	μ	σ	μ	σ
1	1,00	0,00	92,50	0,83
2	1,55	0,04	59,87	1,18
4	3,52	0,16	26,30	1,36
6	6,14	0,07	15,07	0,28
8	7,80	0,20	11,87	0,26
10	9,49	0,11	9,75	0,12
12	10,31	0,26	8,97	0,15
14	10,40	0,76	8,93	0,62
16	11,17	1,33	8,38	1,09
18	5,19	0,19	17,84	0,53
20	5,85	0,25	15,82	0,64
22	6,37	0,36	14,56	0,90
24	6,71	0,24	13,80	0,55
26	7,82	0,71	11,91	1,08
28	9,07	0,60	10,22	0,63
30	8,69	0,15	10,65	0,23
32	7,57	0,25	12,22	0,30

8000^2 células

	Speedup		Tiempo	
Tareas	μ	σ	μ	σ
1	1,00	0,00	363,33	2,50
2	1,54	0,01	235,88	0,54
4	3,64	0,16	99,82	3,79
6	6,72	0,14	54,07	1,29
8	9,14	0,14	39,77	0,46
10	11,10	0,26	32,73	0,71
12	13,12	0,20	27,70	0,40
14	14,86	0,18	24,46	0,18
16	16,38	0,49	22,20	0,59
18	9,41	0,12	38,62	0,50
20	9,45	0,26	38,46	0,81
22	9,42	0,56	38,65	2,07
24	9,99	0,83	36,58	3,34
26	10,68	0,24	34,01	0,63
28	10,38	0,12	35,01	0,29
30	9,77	0,48	37,22	1,70
32	9,58	0,31	37,95	1,29

12000^2 células

	Speedup		Tiempo	
Tareas	μ	σ	μ	σ
1	1,00	0,00	820,60	6,19
2	1,46	0,06	561,41	24,28
4	3,59	0,16	228,52	9,28
6	7,06	0,15	116,22	2,03
8	9,54	0,11	86,07	0,99
10	11,63	0,10	70,56	0,56
12	13,73	0,09	59,78	0,53
14	15,55	0,16	52,79	0,60
16	16,89	0,46	48,62	1,58
18	10,98	0,11	74,72	0,75
20	11,55	0,17	71,05	0,49
22	12,90	0,11	63,64	0,11
24	13,13	0,10	62,50	0,32
26	12,61	0,57	65,16	2,85
28	13,18	0,57	62,35	2,66
30	13,43	0,54	61,19	2,41
32	13,36	0,08	61,44	0,31

Tabla D.1: Tablas descriptivas de las gráficas presentadas en las figuras C.1 y C.2. Se presentan las medias y desviaciones típicas del índice *speedup* y del tiempo medido en segundos para la solución de cerrojo único estándar.

2000^2 células

	Speedup		Tiempo	
Tareas	μ	σ	μ	σ
1	1,00	0,00	24,49	0,10
2	1,58	0,14	15,57	1,31
4	3,19	0,15	7,69	0,32
6	3,71	0,31	6,64	0,62
8	4,24	0,61	5,88	0,87
10	4,93	0,64	5,02	0,57
12	5,11	1,12	5,00	1,30
14	6,68	1,22	3,75	0,60
16	5,88	0,24	4,17	0,15
18	4,09	0,12	6,00	0,18
20	4,59	0,19	5,34	0,21
22	5,88	0,33	4,17	0,21
24	4,66	0,41	5,29	0,45
26	4,56	0,24	5,38	0,30
28	5,24	0,50	4,71	0,45
30	5,50	0,45	4,48	0,39
32	4,93	0,71	5,06	0,82

4000^2 células

	Speedup		Tiempo	
Tareas	μ	σ	μ	σ
1	1,00	0,00	92,47	0,46
2	1,57	0,06	59,05	2,17
4	3,45	0,30	26,93	2,11
6	6,20	0,14	14,93	0,40
8	7,67	0,33	12,07	0,54
10	8,10	1,97	12,13	3,88
12	10,34	0,55	8,97	0,51
14	10,74	0,50	8,63	0,43
16	11,08	1,18	8,42	0,87
18	5,07	0,21	18,28	0,85
20	5,67	0,06	16,31	0,25
22	6,39	0,26	14,47	0,53
24	6,72	0,18	13,78	0,42
26	6,82	0,28	13,57	0,57
28	7,43	0,41	12,47	0,73
30	7,70	0,41	12,04	0,61
32	7,19	0,21	12,88	0,42

8000^2 células

	Speedup		Tiempo	
Tareas	μ	σ	μ	σ
1	1,00	0,00	363,53	2,96
2	1,52	0,02	239,54	0,97
4	3,61	0,07	100,80	1,26
6	6,71	0,20	54,20	1,33
8	9,06	0,11	40,14	0,47
10	11,09	0,19	32,80	0,73
12	13,00	0,40	27,98	0,64
14	14,69	0,20	24,75	0,51
16	15,93	0,67	22,85	1,01
18	9,09	0,21	40,02	0,72
20	9,06	0,28	40,15	0,88
22	9,58	0,14	37,98	0,85
24	9,32	1,18	39,47	5,10
26	10,04	0,89	36,49	3,86
28	10,35	0,20	35,14	0,45
30	10,12	0,53	36,00	1,94
32	9,65	0,16	37,68	0,40

12000^2 células

	Speedup		Tiempo	
Tareas	μ	σ	μ	σ
1	1,00	0,00	816,10	7,98
2	1,46	0,05	559,13	21,48
4	3,35	0,18	244,18	13,22
6	7,01	0,09	116,47	1,36
8	9,48	0,08	86,05	0,35
10	10,13	2,53	85,86	28,13
12	13,72	0,28	59,48	0,66
14	15,48	0,14	52,71	0,26
16	16,57	0,39	49,27	0,85
18	10,46	0,82	78,41	6,48
20	11,36	0,16	71,84	0,41
22	11,16	0,94	73,49	6,19
24	13,00	0,09	62,80	0,17
26	12,40	0,57	65,93	3,11
28	13,25	0,17	61,60	0,24
30	13,46	0,20	60,63	0,42
32	13,12	0,12	62,22	0,13

Tabla D.2: Tablas descriptivas de las gráficas presentadas en las figuras C.3 y C.4. Se presentan las medias y desviaciones típicas del índice *speedup* y del tiempo medido en segundos para la solución de cerrojo único de alto nivel.

2000^2 células

	Speedup		Tiempo	
Tareas	μ	σ	μ	σ
1	1,00	0,00	27,52	0,25
2	1,73	0,11	15,94	0,92
4	3,72	0,37	7,46	0,87
6	5,08	0,22	5,43	0,25
8	4,73	0,68	5,90	0,76
10	7,03	0,51	3,93	0,28
12	7,36	1,56	3,89	0,94
14	7,84	1,44	3,59	0,58
16	6,57	0,93	4,25	0,53
18	5,38	0,33	5,13	0,32
20	5,36	0,50	5,17	0,46
22	6,12	0,21	4,50	0,17
24	5,94	0,40	4,65	0,30
26	5,79	0,31	4,76	0,24
28	6,83	0,53	4,05	0,28
30	6,59	0,29	4,19	0,15
32	6,68	0,92	4,19	0,66

4000^2 células

	Speedup		Tiempo	
Tareas	μ	σ	μ	σ
1	1,00	0,00	105,61	0,82
2	1,77	0,04	59,81	0,93
4	4,58	0,25	23,12	1,19
6	7,08	0,51	14,97	1,10
8	9,24	0,32	11,43	0,34
10	10,64	0,41	9,94	0,37
12	10,96	1,75	9,86	1,79
14	12,43	0,87	8,53	0,65
16	12,76	1,30	8,35	0,96
18	5,99	0,15	17,63	0,51
20	6,89	0,41	15,36	0,86
22	7,75	0,14	13,64	0,24
24	8,10	0,56	13,09	0,90
26	9,06	0,43	11,67	0,56
28	10,42	0,95	10,21	1,00
30	9,74	1,52	11,06	1,87
32	8,63	0,44	12,26	0,66

8000^2 células

	Speedup		Tiempo	
Tareas	μ	σ	μ	σ
1	1,00	0,00	413,73	3,23
2	1,74	0,01	237,85	1,79
4	5,04	0,32	82,32	5,35
6	7,52	0,07	54,97	0,39
8	10,42	0,39	39,73	1,19
10	12,45	0,15	33,23	0,46
12	14,89	0,19	27,79	0,45
14	16,78	0,13	24,66	0,30
16	16,22	3,06	26,38	6,16
18	10,24	0,76	40,59	3,39
20	10,73	0,17	38,56	0,49
22	10,80	0,14	38,31	0,40
24	11,94	0,38	34,68	0,85
26	12,07	0,21	34,28	0,50
28	11,75	0,18	35,21	0,51
30	11,47	0,68	36,17	2,22
32	11,20	0,08	36,95	0,06

12000^2 células

	Speedup		Tiempo	
Tareas	μ	σ	μ	σ
1	1,00	0,00	978,94	3,94
2	1,65	0,01	593,33	4,39
4	5,21	0,08	188,05	3,00
6	7,78	1,29	129,10	25,91
8	10,48	1,72	95,72	18,94
10	12,83	2,06	78,09	14,98
12	15,38	2,41	65,11	12,20
14	15,23	3,92	67,97	19,37
16	19,33	2,55	51,41	7,80
18	12,90	0,69	76,09	4,36
20	13,63	0,19	71,82	1,03
22	15,21	0,36	64,38	1,60
24	15,40	0,62	63,67	2,75
26	14,79	0,73	66,31	3,10
28	15,65	0,52	62,59	2,21
30	16,05	0,43	61,05	1,73
32	15,74	0,25	62,22	0,98

Tabla D.3: Tablas descriptivas de las gráficas presentadas en las figuras C.5 y C.6. Se presentan las medias y desviaciones típicas del índice *speedup* y del tiempo medido en segundos para la solución de cerrojo estándar por partición.

2000^2 células

Tareas	Speedup μ	Speedup σ	Tiempo μ	Tiempo σ
1	1,00	0,00	27,60	0,08
2	1,71	0,05	16,13	0,52
4	4,24	0,32	6,53	0,48
6	5,19	0,15	5,32	0,15
8	6,39	0,24	4,33	0,17
10	7,14	0,60	3,88	0,35
12	8,90	0,26	3,10	0,09
14	9,69	0,29	2,85	0,08
16	10,59	0,52	2,61	0,13
18	5,37	0,27	5,15	0,27
20	5,99	0,43	4,63	0,36
22	6,94	0,69	4,00	0,39
24	7,21	0,76	3,85	0,38
26	6,75	0,59	4,11	0,35
28	7,04	2,51	4,42	1,94
30	7,61	0,99	3,67	0,43
32	7,33	0,54	3,78	0,31

4000^2 células

Tareas	Speedup μ	Speedup σ	Tiempo μ	Tiempo σ
1	1,00	0,00	107,22	0,36
2	1,84	0,04	58,40	1,25
4	4,42	0,23	24,27	1,25
6	7,11	0,36	15,11	0,73
8	8,53	1,66	13,02	3,12
10	11,00	0,15	9,75	0,13
12	11,98	0,35	8,96	0,29
14	12,78	0,06	8,39	0,03
16	13,33	0,99	8,08	0,66
18	6,18	0,08	17,35	0,23
20	6,94	0,15	15,46	0,35
22	7,81	0,10	13,74	0,20
24	8,59	0,26	12,50	0,36
26	9,52	0,16	11,26	0,21
28	9,85	0,79	10,93	0,89
30	10,84	0,66	9,92	0,58
32	9,23	0,43	11,63	0,57

8000^2 células

Tareas	Speedup μ	Speedup σ	Tiempo μ	Tiempo σ
1	1,00	0,00	415,76	3,65
2	1,77	0,03	234,57	1,61
4	4,87	0,26	85,59	4,69
6	7,66	0,12	54,26	0,97
8	10,25	0,38	40,61	1,33
10	12,76	0,27	32,60	0,66
12	15,23	0,27	27,31	0,26
14	16,84	0,60	24,71	0,69
16	18,10	0,69	23,00	0,94
18	10,86	0,33	38,32	1,23
20	10,61	0,89	39,43	3,60
22	10,55	0,72	39,56	2,73
24	12,11	0,25	34,36	0,80
26	11,45	1,40	36,84	5,49
28	10,55	0,94	39,61	3,41
30	11,74	0,45	35,45	1,07
32	10,99	0,72	37,96	2,47

12000^2 células

Tareas	Speedup μ	Speedup σ	Tiempo μ	Tiempo σ
1	1,00	0,00	981,21	12,71
2	1,66	0,03	590,04	1,68
4	4,76	1,07	217,20	66,11
6	7,45	1,89	141,22	50,05
8	9,44	2,64	111,57	36,67
10	13,08	1,51	75,85	9,72
12	13,91	3,61	74,79	21,94
14	17,82	1,81	55,53	6,20
16	17,67	3,93	57,95	14,56
18	12,63	0,96	78,03	6,00
20	13,78	0,25	71,25	1,11
22	15,37	0,30	63,85	0,78
24	15,63	0,31	62,78	0,96
26	15,35	0,27	63,94	0,72
28	15,69	0,78	62,67	3,09
30	16,29	0,31	60,25	0,71
32	15,87	0,27	61,81	0,51

Tabla D.4: Tablas descriptivas de las gráficas presentadas en las figuras C.7 y C.8. Se presentan las medias y desviaciones típicas del índice *speedup* y del tiempo medido en segundos para la solución de cerrojo de alto nivel por partición.

2000^2 células

	Speedup		Tiempo	
Tareas	μ	σ	μ	σ
1	1,00	0,00	24,27	0,42
2	1,56	0,06	15,56	0,46
4	3,26	0,48	7,57	1,05
6	4,15	0,79	6,02	1,21
8	4,72	0,95	5,29	1,00
10	5,35	1,46	4,80	1,35
12	6,56	0,76	3,73	0,35
14	6,80	2,01	3,80	1,06
16	6,65	0,63	3,68	0,37
18	4,58	0,75	5,43	0,98
20	4,73	0,17	5,13	0,20
22	5,08	0,48	4,82	0,54
24	4,93	0,39	4,94	0,32
26	4,59	0,62	5,36	0,70
28	5,54	0,63	4,42	0,42
30	5,39	0,20	4,51	0,19
32	6,22	0,34	3,91	0,24

4000^2 células

	Speedup		Tiempo	
Tareas	μ	σ	μ	σ
1	1,00	0,00	93,97	0,59
2	1,46	0,23	65,91	12,45
4	3,43	0,27	27,55	2,04
6	6,06	0,23	15,53	0,66
8	8,05	0,22	11,68	0,27
10	9,49	0,35	9,91	0,38
12	10,37	0,39	9,07	0,31
14	10,86	0,56	8,67	0,40
16	10,43	0,29	9,02	0,29
18	5,43	0,11	17,30	0,25
20	6,08	0,30	15,49	0,69
22	6,61	0,15	14,21	0,29
24	7,07	0,47	13,34	0,96
26	7,97	0,44	11,81	0,57
28	9,33	0,73	10,13	0,86
30	8,85	0,54	10,65	0,70
32	7,39	0,52	12,76	0,83

8000^2 células

	Speedup		Tiempo	
Tareas	μ	σ	μ	σ
1	1,00	0,00	367,82	5,60
2	1,57	0,02	234,26	2,36
4	3,72	0,12	98,99	2,29
6	6,78	0,16	54,27	1,18
8	9,28	0,16	39,66	0,72
10	11,30	0,37	32,58	1,10
12	11,99	2,84	32,37	9,51
14	14,96	0,46	24,59	0,41
16	16,42	0,18	22,40	0,34
18	9,39	0,48	39,21	1,61
20	9,47	0,40	38,87	1,47
22	9,37	0,53	39,32	1,76
24	10,67	0,21	34,47	0,51
26	10,76	0,05	34,19	0,39
28	9,90	0,91	37,38	3,36
30	10,59	0,15	34,74	0,32
32	9,84	0,10	37,37	0,22

12000^2 células

	Speedup		Tiempo	
Tareas	μ	σ	μ	σ
1	1,00	0,00	816,88	10,75
2	1,44	0,07	569,06	26,51
4	3,41	0,12	239,65	9,51
6	7,16	0,06	114,19	1,23
8	9,48	0,23	86,20	1,29
10	11,57	0,18	70,63	0,80
12	13,70	0,26	59,62	0,43
14	14,01	3,40	61,77	19,09
16	17,25	0,51	47,37	0,94
18	11,01	0,18	74,20	0,68
20	11,44	0,12	71,40	0,29
22	12,78	0,17	63,91	0,23
24	13,06	0,19	62,55	0,17
26	12,56	0,43	65,11	2,74
28	13,36	0,19	61,14	0,09
30	13,35	0,63	61,29	2,39
32	13,32	0,21	61,33	0,18

Tabla D.5: Tablas descriptivas de las gráficas presentadas en las figuras C.9 y C.10. Se presentan las medias y desviaciones típicas del índice *speedup* y del tiempo medido en segundos para la solución de cerrojo estándar para zonas fronterizas.

2000^2 células

Tareas	Speedup μ	Speedup σ	Tiempo μ	Tiempo σ
1	1,00	0,00	24,35	0,11
2	1,44	0,03	16,88	0,32
4	3,13	0,38	7,88	0,97
6	3,85	0,47	6,40	0,72
8	5,31	0,27	4,59	0,25
10	5,07	1,40	5,21	1,95
12	6,14	1,05	4,06	0,77
14	5,75	0,56	4,26	0,40
16	5,41	0,83	4,57	0,63
18	4,61	0,12	5,29	0,15
20	4,34	0,49	5,66	0,68
22	5,54	0,33	4,40	0,26
24	4,61	0,46	5,33	0,49
26	4,79	0,68	5,18	0,86
28	4,94	0,28	4,94	0,28
30	4,91	0,37	4,97	0,35
32	6,29	0,18	3,88	0,09

4000^2 células

Tareas	Speedup μ	Speedup σ	Tiempo μ	Tiempo σ
1	1,00	0,00	92,17	0,53
2	1,46	0,01	62,88	0,54
4	3,26	0,17	28,29	1,40
6	5,96	0,20	15,47	0,56
8	7,43	0,35	12,42	0,56
10	9,13	0,18	10,09	0,25
12	10,06	0,55	9,18	0,49
14	10,20	0,38	9,04	0,32
16	11,24	0,46	8,21	0,34
18	4,77	0,10	19,33	0,38
20	5,74	0,14	16,05	0,41
22	6,37	0,31	14,49	0,65
24	6,59	0,33	14,01	0,68
26	7,49	0,58	12,36	1,00
28	8,70	0,55	10,63	0,67
30	8,77	0,80	10,59	1,01
32	7,59	0,43	12,18	0,72

8000^2 células

Tareas	Speedup μ	Speedup σ	Tiempo μ	Tiempo σ
1	1,00	0,00	363,33	1,97
2	1,39	0,15	264,19	31,30
4	3,46	0,24	105,53	8,12
6	6,51	0,06	55,83	0,72
8	8,73	0,14	41,62	0,54
10	10,52	0,39	34,56	1,15
12	12,41	0,14	29,29	0,24
14	14,17	0,08	25,64	0,04
16	15,71	0,12	23,13	0,23
18	8,06	0,14	45,12	0,60
20	9,16	0,13	39,66	0,59
22	8,67	0,48	41,98	2,20
24	9,99	0,37	36,39	1,32
26	10,12	0,20	35,91	0,68
28	9,70	0,54	37,53	2,01
30	9,78	0,30	37,19	1,33
32	9,61	0,16	37,83	0,45

12000^2 células

Tareas	Speedup μ	Speedup σ	Tiempo μ	Tiempo σ
1	1,00	0,00	809,65	3,77
2	1,37	0,01	590,14	1,66
4	3,04	0,54	273,47	58,75
6	6,56	0,13	123,50	2,34
8	7,93	1,90	108,26	33,98
10	10,79	0,12	75,02	0,52
12	12,82	0,06	63,14	0,03
14	14,59	0,08	55,50	0,20
16	13,32	3,67	64,43	17,55
18	9,89	0,15	81,84	0,86
20	10,61	0,46	76,38	3,34
22	11,83	0,08	68,41	0,52
24	12,54	0,04	64,58	0,21
26	12,14	0,06	66,72	0,12
28	12,45	0,10	65,02	0,38
30	12,86	0,06	62,98	0,31
32	12,61	0,06	64,23	0,14

Tabla D.6: Tablas descriptivas de las gráficas presentadas en las figuras C.11 y C.12. Se presentan las medias y desviaciones típicas del índice *speedup* y del tiempo medido en segundos para la solución de cerrojo de alto nivel para zonas fronterizas.

1000 generaciones

	Speedup		Tiempo	
Tareas	μ	σ	μ	σ
1	1,00	0,00	2,73	0,09
2	0,41	0,05	6,77	0,82
4	0,62	0,12	4,53	1,02
6	0,69	0,09	4,00	0,41
8	1,00	0,12	2,76	0,36
10	1,10	0,25	2,58	0,59
12	3,82	0,53	0,73	0,13
14	1,16	0,46	2,57	0,77
16	1,38	0,35	2,09	0,55
18	1,53	0,34	1,84	0,35
20	1,53	0,10	1,79	0,16
22	1,67	0,16	1,65	0,20
24	1,96	0,39	1,45	0,37
26	2,01	0,25	1,38	0,20
28	1,92	0,34	1,46	0,30
30	1,91	0,32	1,45	0,18
32	2,85	0,97	1,05	0,37

2000 generaciones

	Speedup		Tiempo	
Tareas	μ	σ	μ	σ
1	1,00	0,00	19,22	0,14
2	0,42	0,14	51,12	18,10
4	0,58	0,11	34,08	6,38
6	0,72	0,07	26,76	2,66
8	2,01	0,36	9,84	2,14
10	2,69	0,59	7,45	1,98
12	3,61	0,96	5,66	1,64
14	4,52	0,87	4,40	0,98
16	5,23	1,50	4,00	1,55
18	3,95	1,08	5,27	1,98
20	4,89	1,02	4,05	0,84
22	6,90	0,67	2,81	0,29
24	7,23	0,33	2,66	0,14
26	6,04	2,97	4,80	4,45
28	1,75	0,32	11,25	2,07
30	2,23	0,09	8,65	0,41
32	2,14	0,19	9,03	0,73

4000 generaciones

	Speedup		Tiempo	
Tareas	μ	σ	μ	σ
1	1,00	0,00	150,50	1,23
2	1,10	0,19	140,61	29,57
4	3,31	0,28	45,69	3,42
6	3,41	0,28	44,32	3,62
8	3,69	0,44	41,23	5,14
10	6,48	3,64	48,52	60,51
12	7,96	0,77	19,04	1,99
14	5,77	2,47	34,26	25,64
16	9,06	2,36	17,38	3,98
18	5,36	0,29	28,13	1,73
20	4,73	0,53	32,18	4,13
22	5,59	1,74	29,20	10,10
24	8,33	0,54	18,11	1,05
26	7,59	0,54	19,92	1,64
28	6,27	2,78	27,67	11,68
30	10,24	0,90	14,79	1,30
32	7,89	1,47	19,65	4,19

17500 generaciones

	Speedup		Tiempo	
Tareas	μ	σ	μ	σ
1	1,00	0,00	10913,65	470,28
2	1,88	0,01	5786,25	270,53
4	3,78	0,14	2892,19	173,73
6	5,82	0,18	1879,48	136,02
8	7,72	0,32	1416,22	92,89
10	5,00	3,42	3687,50	3237,43
12	6,78	3,76	2109,88	1273,00
14	12,48	1,33	886,25	143,52
16	9,94	5,49	1405,82	746,42
18	9,62	0,61	1140,49	129,40
20	8,77	3,92	1699,05	1355,42
22	9,05	4,18	1697,70	1427,18
24	8,14	4,78	1894,53	1334,47
26	11,13	1,66	999,34	175,53
28	11,05	3,90	1143,66	590,06
30	6,54	4,03	2099,32	968,25
32	10,85	4,40	1263,81	849,50

Tabla D.7: Tablas descriptivas de las gráficas presentadas en las figuras C.13 y C.14. Se presentan las medias y desviaciones típicas del índice *speedup* y del tiempo medido en segundos para la solución de cerrojo estándar para zonas fronterizas, con dominio de cómputo creciente.

1000 generaciones

Tareas	Speedup μ	Speedup σ	Tiempo μ	Tiempo σ
1	1,00	0,00	2,70	0,13
2	0,39	0,06	6,99	1,14
4	0,68	0,07	3,96	0,41
6	0,79	0,04	3,40	0,16
8	1,08	0,26	2,59	0,54
10	1,04	0,14	2,61	0,25
12	2,93	1,44	1,17	0,70
14	1,33	0,33	2,13	0,61
16	1,39	0,23	1,98	0,35
18	1,47	0,30	1,91	0,50
20	1,52	0,27	1,81	0,32
22	1,45	0,13	1,87	0,19
24	1,79	0,34	1,56	0,37
26	2,10	0,20	1,29	0,17
28	2,01	0,39	1,37	0,19
30	1,99	0,15	1,36	0,12
32	2,74	0,72	1,04	0,29

2000 generaciones

Tareas	Speedup μ	Speedup σ	Tiempo μ	Tiempo σ
1	1,00	0,00	19,54	0,43
2	0,42	0,15	51,38	16,28
4	0,59	0,05	33,21	2,49
6	0,69	0,06	28,55	2,36
8	2,10	0,48	9,66	1,98
10	2,96	0,39	6,70	0,91
12	3,66	0,48	5,40	0,72
14	4,74	1,15	4,32	1,13
16	5,26	0,96	3,81	0,71
18	4,31	0,39	4,56	0,43
20	5,72	0,29	3,42	0,17
22	6,21	0,86	3,19	0,37
24	4,05	2,30	6,35	3,61
26	4,38	3,52	8,74	7,16
28	2,60	1,97	10,07	4,71
30	2,04	0,27	9,70	1,44
32	2,19	0,26	9,01	0,85

4000 generaciones

Tareas	Speedup μ	Speedup σ	Tiempo μ	Tiempo σ
1	1,00	0,00	149,38	1,85
2	1,23	0,41	129,18	33,79
4	2,85	0,10	52,53	1,50
6	2,45	1,24	76,38	40,94
8	3,63	0,30	41,31	3,55
10	5,88	2,89	32,59	20,55
12	8,17	1,09	18,52	2,34
14	7,37	0,78	20,46	2,23
16	6,66	4,08	30,46	20,10
18	4,88	0,66	31,06	4,64
20	5,16	1,12	29,91	5,95
22	5,64	0,46	26,65	2,49
24	8,07	1,16	18,80	2,78
26	8,11	1,39	18,81	3,09
28	6,47	1,29	23,78	4,74
30	9,06	1,50	16,85	2,90
32	5,08	2,61	36,66	18,89

17500 generaciones

Tareas	Speedup μ	Speedup σ	Tiempo μ	Tiempo σ
1	1,00	0,00	10961,75	556,00
2	1,82	0,03	6020,32	214,47
4	3,06	1,16	4423,79	2956,44
6	5,19	0,36	2120,18	166,09
8	5,60	2,56	2541,40	1764,50
10	7,23	3,15	1953,37	1355,54
12	8,21	2,83	1517,18	720,55
14	6,87	5,28	2916,08	2291,49
16	8,25	6,86	2677,74	2168,92
18	6,86	3,51	2104,64	1356,71
20	10,35	0,27	1059,99	76,54
22	8,53	3,78	1518,54	703,81
24	12,01	0,67	914,76	61,00
26	7,60	4,00	1776,79	887,02
28	11,02	3,25	1087,09	417,31
30	10,50	3,17	1152,99	481,33
32	9,10	3,55	1348,00	493,07

Tabla D.8: Tablas descriptivas de las gráficas presentadas en las figuras C.15 y C.16. Se presentan las medias y desviaciones típicas del índice *speedup* y del tiempo medido en segundos para la solución de cerrojo de alto nivel para zonas fronterizas, con dominio de cómputo creciente.

Apéndice E

Manual de Usuario

Se incluye en este apartado una guía básica de uso de nuestra aplicación.

E.1. Introducción

TumoralGrowth es un simulador de crecimiento neoplásico basado en autómatas celulares desarrollado en Java. El simulador hará uso de toda la capacidad de cómputo de su máquina, por lo que se recomienda que cierre el resto de aplicaciones si quiere obtener el mayor rendimiento posible.

Cabe destacar el perfil multiplataforma de nuestra aplicación y el alto nivel de personalización mediante los parámetros de ejecución del autómata. Así mismo, se proporciona información detallada, como las acciones de las células (mediante el código de colores), el crecimiento y el nivel de entropía de la población.

E.2. Instalación

No se precisa instalación del programa, sólo del entorno de ejecución de Java. La información sobre la instalación de éste puede verse en la documentación oficial del desarrollador.

Los requisitos de instalación mínimos de hardware y software de este sistema son los requisitos mínimos establecidos por Oracle en la versión 8 del entorno de ejecución de Java.

No obstante, se recomienda una máquina capaz de soportar sesiones intensivas de cómputo, con capacidad de RAM suficiente para albergar dominios tisulares de un tamaño relevante (por ejemplo, 12000x12000) y un procesador con más de un *core*.

Como ejemplo, la máquina con la que se ha trabajado y realizado el simulador dispone de 4 núcleos y 8GB de RAM.

E.3. Uso del Sistema

Para iniciar el simulador, simplemente haga doble clic sobre el fichero `.jar` proporcionado. Si por algún motivo su sistema no abriera correctamente el paquete Java, puede usted ejecutarlo manualmente situándose en el directorio donde se encuentra y ejecutando el siguiente comando:

```
java -jar TumoralGrowth.java
```

Al iniciarse, podrá ver la siguiente ventana de la figura E.1.

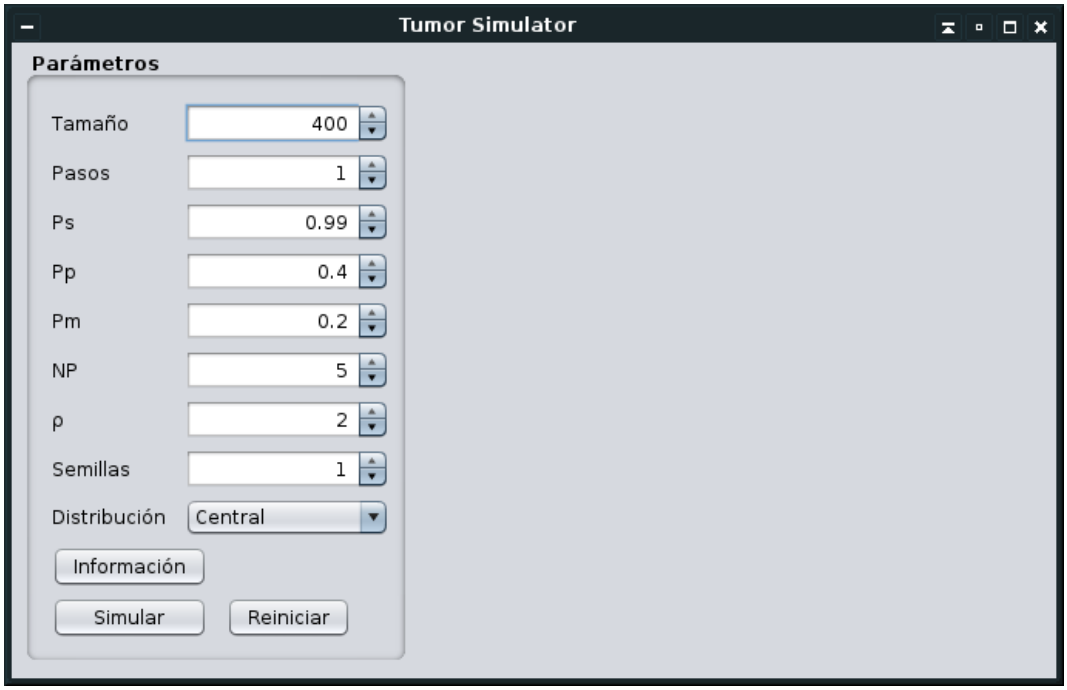

Figura E.1: Ventana principal de la aplicación

A la izquierda podrá ver la columna de configuración de parámetros.

- *Ps* es la probabilidad de superviviencia celular. Está acotada en el intervalo $[0, 1]$.
- *Pp* es la probabilidad de proliferación. Está acotada en el intervalo $[0, 1]$.
- *Pm* es la probabilidad de migración. Está acotada en el intervalo $[0, 1]$.
- *NP* es el número de señales de proliferación necesarias para llevar a cabo la mitosis. Es un entero no negativo.
- ρ es el número máximo de veces que una célula puede proliferar sin morir. Es un entero no negativo. Si se especifica 0, las células son contempladas como *stem* (células que pueden proliferar infinitamente sin morir).
- *Semillas* especifica el número inicial de células *stem*. Ha de ser, como mínimo, 1.
- La *distribución* especifica la forma de ordenar las células *stem* en el dominio tisular.
  - *Central* posiciona las células en el centro del dominio tisular, en un cuadro de tamaño $(\sqrt{s})$x$(\sqrt{s})$, donde s es el número de *semillas*.
  - *Aleatoria* posiciona las células siguiendo una distribución aleatoria uniforme en todo el dominio tisular.

- *Tamaño* es el número de células del lado del cuadrado que representa el dominio tisular. Será, como mínimo, de $\lceil\sqrt{s}\rceil$, siendo s el número de *semillas*.

- *Pasos* define el número de pasos discretos de tiempo que se ejecutarán cada vez que se vaya a actualizar la interfaz.

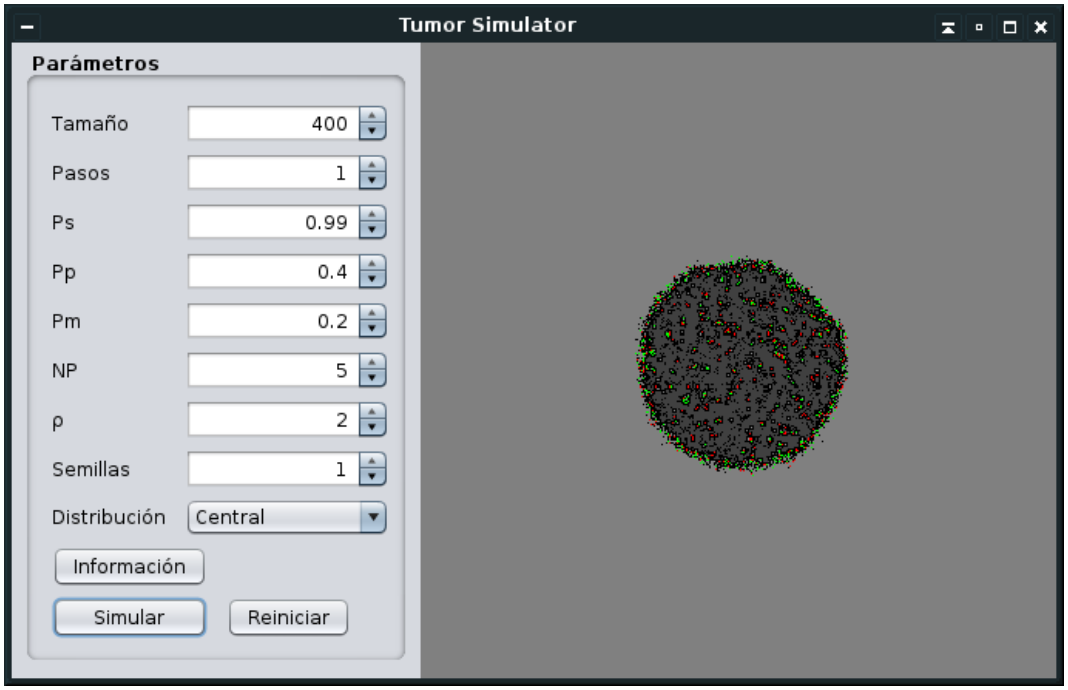

Figura E.2: Ventana principal de la aplicación con una simulación ya iniciada.

En la parte inferior de la columna se encuentran tres botones.

- *Simular* es un botón de dos estados.

  - Mientras su estado sea *activo*, el autómata se estará ejecutando y la interfaz será actualizada cada vez que se tenga un nuevo estado (figura E.2).

  - Mientras su estado sea *inactivo*, el autómata estará pausado y la interfaz permanecerá estática.

- *Reiniciar* establecerá el autómata celular como si acabara de ser creado, y emplazará las semillas *stem* iniciales.

- *Información* abrirá la ventana de información adicional de la figura E.3, donde se pueden observar parámetros como el número de generaciones, la población, y las gráficas de crecimiento y entropía. Si la ventana ya estuviese abierta, la traería a la primera capa (encima de la otra ventana) y obtendría el foco.

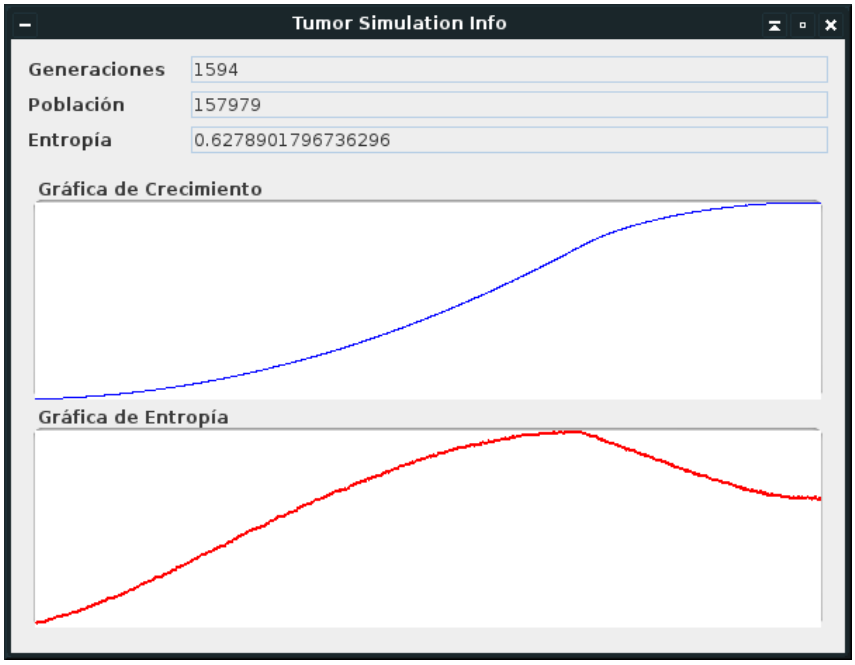

Figura E.3: Ventana de información adicional.

E.4. Mediciones desde la Línea de Comandos

Pueden realizarse tomas de tiempo y cálculos de *speedup* desde la propia línea de comandos, tanto para la versión Java como para la versión en C++.

E.4.1. Línea de Comandos: Java

Pueden realizarse mediciones de *speedup* mediante el siguiente comando:

```
java -cp TumoralGrowth.jar tumoralgrowthautomaton.Speedup <size>
    <maxTasks> <stepsBetweenTasks> <generations>
```

donde los argumentos de la linea de comandos se corresponden con

- `size` especifica las dimensiones del lado del dominio tisular.
- `maxTasks` especifica el número máximo de tareas que se ejecutarán.
- `stepsBetweenTasks` establece de cuánto en cuánto se incrementará el número de tareas. Ejemplo: si se especifica 2, se ejecutará para $1, 2, 4, ..., n$ siendo $n \in \mathbb{Z} : \exists q \in \mathbb{Z} : n = 2q \wedge 2q \leq \text{maxTareas} \wedge 2(q+1) > \text{maxTareas}$.
- `generations` establece el número de generaciones del autómata que se computarán.

De la misma forma, pueden tomarse tiempos puntuales mediante el uso del siguiente comando:

```
java -cp TumoralGrowth.jar tumoralgrowthautomaton.Time <size>
    <generations> <tasks>
```

donde los parámetros se corresponden con

- `size` establece el tamaño del lado del dominio tisular.
- `generations` especifica el número de generaciones del autómata que se calcularán.
- `tasks` establece el número de tareas que se usarán para ello.

E.4.2. Línea de Comandos: C++

Ejecute la orden `make` para compilar el programa. Una vez hecho, puede tomar mediciones de *speedup* mediante el siguiente comando:

```
./speedup <size> <maxTasks> <stepsBetweenTasks> <generations>
```

donde los argumentos de la linea de comandos se corresponden con

- `size` especifica las dimensiones del lado del dominio tisular.
- `maxTasks` especifica el número máximo de tareas que se ejecutarán.
- `stepsBetweenTasks` establece de cuánto en cuánto se incrementará el número de tareas. Ejemplo: si se especifica 2, se ejecutará para $1, 2, 4, ..., n$ siendo $n \in \mathbb{Z} : \exists q \in \mathbb{Z} : n = 2q \wedge 2q \leq \text{maxTareas} \wedge 2(q+1) > \text{maxTareas}$.
- `generations` establece el número de generaciones del autómata que se computarán.

De la misma forma, pueden tomarse tiempos puntuales mediante el uso del siguiente comando:

```
./time <size> <generations> <tasks>
```

donde los parámetros se corresponden con

- `size` establece el tamaño del lado del dominio tisular.
- `generations` especifica el número de generaciones del autómata que se calcularán.
- `tasks` establece el número de tareas que se usarán para ello.

Bibliografía

[1] T. Haigh, M. Priestley y C. Rope. «Los Alamos Bets on ENIAC: Nuclear Monte Carlo Simulations, 1947-1948». En: *IEEE Annals of the History of Computing* 36.3 (jul. de 2014), págs. 42-63. ISSN: 1058-6180. DOI: 10.1109/MAHC.2014.40.

[2] Peter Bright. *Intel confirms tick-tock-shattering Kaby Lake processor as Moore's Law falters*. Jul. de 2015. URL: http://arstechnica.com/gadgets/2015/07/intel-confirms-tick-tock-shattering-kaby-lake-processor-as-moores-law-falters/ (visitado 29-05-2016).

[3] Dustin D. Phan y J. S. Lowengrub. «A discrete cellular automaton model demonstrates cell motility increases fitness in solid tumors». En: *The UCI Undergraduate Research Journal* (2010), págs. 55-66. URL: http://www.urop.uci.edu/journal/journal09/06_phan.pdf.

[4] Dominik Wodarz y Natalia Komarova. *Computational Biology Of Cancer: Lecture Notes And Mathematical Modeling*. {World Scientific Publishing Company}, ene. de 2005. ISBN: 978-981-256-027-8.

[5] Philippe Tracqui. «From passive diffusion to active cellular migration in mathematical models of tumour invasion». en. En: *Acta Biotheoretica* 43.4 (dic. de 1995), págs. 443-464. ISSN: 0001-5342, 1572-8358. DOI: 10.1007/BF00713564. URL: http://link.springer.com/article/10.1007/BF00713564.

[6] J. P. Ward y J. R. King. «Mathematical modelling of avascular-tumour growth». en. En: *Mathematical Medicine and Biology* 14.1 (mar. de 1997), págs. 39-69. ISSN: 1477-8599, 1477-8602. DOI: 10.1093/imammb/14.1.39.

[7] J. P. Ward y J. R. King. «Mathematical modelling of avascular-tumour growth II: Modelling growth saturation». en. En: *Mathematical Medicine and Biology* 16.2 (jun. de 1999), págs. 171-211. ISSN: 1477-8599, 1477-8602. DOI: 10.1093/imammb/16.2.171. URL: http://imammb.oxfordjournals.org/content/16/2/171.

[8] Alexander R. A. Anderson. «A hybrid mathematical model of solid tumour invasion: the importance of cell adhesion». en. En: *Mathematical Medicine and Biology* 22.2 (jun. de 2005), págs. 163-186. ISSN: 1477-8599, 1477-8602. DOI: 10.1093/imammb/dqi005. URL: http://imammb.oxfordjournals.org/content/22/2/163.

[9] Aalpen A. Patel y col. «A Cellular Automaton Model of Early Tumor Growth and Invasion: The Effects of Native Tissue Vascularity and Increased Anaerobic Tumor Metabolism». En: *Journal of Theoretical Biology* 213.3 (dic. de 2001), págs. 315-331. ISSN: 0022-5193. DOI: 10.1006/jtbi.2001.2385.

[10] A. R. Kansal y col. «Simulated Brain Tumor Growth Dynamics Using a Three-Dimensional Cellular Automaton». En: *Journal of Theoretical Biology* 203.4 (abr. de 2000), págs. 367-382. ISSN: 0022-5193. DOI: 10.1006/jtbi.2000.2000.

[11] Sabine Dormann y Andreas Deutsch. «Modeling of Self-Organized Avascular Tumor Growth with a Hybrid Cellular Automaton». En: *In Silico Biology* 2.3 (sep. de 2002), pág. 393. ISSN: 13866338. URL: http://search.ebscohost.com/login.aspx?direct=true&db=cxh&AN=8769682&site=ehost-live.

[12] S Bandini, G Mauri y R Serra. «Cellular automata: From a theoretical parallel computational model to its application to complex systems». En: *Parallel Computing*. Cellular automata: From modeling to applications 27.5 (abr. de 2001), págs. 539-553. ISSN: 0167-8191. DOI: 10.1016/S0167-8191(00)00076-4.

[13] Antonio J. Tomeu, Alberto G. Salguero y Manuel I. Capel. «Speeding Up Tumor Growth Simulation Using Parallel Programming with Cellular Automata». En: *IEEE-AL (to appear on)* (2016).

[14] E. Kim, T. Shen y X. Huang. «A parallel cellular automata with label priors for interactive brain tumor segmentation». En: *2010 IEEE 23rd International Symposium on Computer-Based Medical Systems (CBMS)*. Oct. de 2010, págs. 232-237. DOI: 10.1109/CBMS.2010.6042647.

[15] Scott Oaks Wong Henry. *Java Threads*. ISBN: 978-0-596-00782-9.

[16] *TIOBE Index | Tiobe - The Software Quality Company*. URL: http://www.tiobe.com/tiobe_index (visitado 31-05-2016).

[17] Anthony Williams. *C++ Concurrency in Action: Practical Multithreading*. en. Manning, 2012. ISBN: 978-1-933988-77-1.

[18] *Current Status : Standard C++*. URL: https://isocpp.org/std/status (visitado 14-06-2016).

[19] Andrew S. Tanenbaum. *Modern operating systems*. Fourth edition. Boston: Pearson, 2015. ISBN: 978-0-13-359162-0.

[20] *Understanding Cancer Research Study Design and How to Evaluate Results*. Jun. de 2013. URL: http://www.cancer.net/research-and-advocacy/introduction-cancer-research/understanding-cancer-research-study-design-and-how-evaluate-results (visitado 31-05-2016).

[21] Dirk Drasdo, Stefan Hoehme y Michael Block. «On the Role of Physics in the Growth and Pattern Formation of Multi-Cellular Systems: What can we Learn from Individual-Cell Based Models?» en. En: *Journal of Statistical Physics* 128.1-2 (abr. de 2007), págs. 287-345. ISSN: 0022-4715, 1572-9613. DOI: 10.1007/s10955-007-9289-x.

[22] Antonio J. Tomeu. «Síntesis del modelo de autómatas celulares en protocolos criptográficos de cifrado en flujo y orientado a redes de feistel». En: *Ed. Proquest* (2002).

[23] *Cellular Automaton Modeling of Biological Pattern Formation*. en. Modeling and Simulation in Science, Engineering and Technology. Boston, MA: Birkhäuser Boston, 2005. ISBN: 978-0-8176-4281-5.

[24] Dario Ponti y col. «Isolation and in vitro propagation of tumorigenic breast cancer cells with stem/progenitor cell properties». En: *Cancer research* 65.13 (2005), págs. 5506-5511.

[25] Heiko Enderling y col. «Paradoxical Dependencies of Tumor Dormancy and Progression on Basic Cell Kinetics». en. En: *Cancer Research* 69.22 (nov. de 2009), págs. 8814-8821. ISSN: 0008-5472, 1538-7445. DOI: 10.1158/0008-5472.CAN-09-2115.

[26] Charles P. Winsor. «The Gompertz curve as a growth curve». En: *Proceedings of the national academy of sciences* 18.1 (1932), págs. 1-8. URL: http://www.pnas.org/content/18/1/1.short.

[27] C. E. Shannon. «A Mathematical Theory of Communication». En: *Tech* 27 (1948), pág. 623.

[28] Kent Beck. *Extreme Programming Explained: Embrace Change*. en. Addison-Wesley Professional, 2000. ISBN: 978-0-201-61641-5.

[29] Venkat Subramaniam. *Programming Concurrency on the JVM: Mastering Synchronization, STM, and Actors*. Inglés. Edición: 1. Dallas, Tex: Pragmatic Bookshelf, sep. de 2011. ISBN: 978-1-934356-76-0.

[30] Enrique Alba. «Evolutionary ComputationParallel evolutionary algorithms can achieve super-linear performance». En: *Information Processing Letters* 82.1 (abr. de 2002), págs. 7-13. ISSN: 0020-0190. DOI: 10.1016/S0020-0190(01)00281-2. URL: http://www.sciencedirect.com/science/article/pii/S0020019001002812.

[31] V. Nageshwara Rao y Vipin Kumar. «Superlinear speedup in parallel state-space search». en. En: *Foundations of Software Technology and Theoretical Computer Science*. Ed. por Kesav V. Nori y Sanjeev Kumar. Lecture Notes in Computer Science 338. DOI: 10.1007/3-540-50517-2_79. Springer Berlin Heidelberg, dic. de 1988, págs. 161-174. ISBN: 978-3-540-50517-4.

[32] Göran Angelo Kaldéren. «A comparative analysis between parallel models in C/C++ and C#/Java: A quantitative comparison between different programming models on how they implement parallelism». En: (2013). URL: http://www.diva-portal.org/smash/record.jsf?pid=diva2:648395 (visitado 09-06-2016).

[33] Antonio J. Tomeu, Alberto G. Salguero y Manuel I. Capel. «A Parallelisation Tale of Two Languages». En: *Annals of Multicore and GPU Programming* (issue n. 2, 2015), págs. 81-94. URL: http://revistaseug.ugr.es/index.php/amgp/article/view/3912.

[34] A. Tomeu, A. G. Salguero y M. Capel. «El API de Concurrencia/Paralelismo en C++. Una Comparativa con Java y C# en Problemas con Coeficiente de Bloqueo Nulo y No Nulo.» En: *Proceedings de las XXIII Jornadas de Concurrencia y Sistemas Distribuidos* (2015), págs. 68-83.

[35] *Comentarios en el kernel de linux - Linux kernel stable tree*. URL: https://git.kernel.org/cgit/linux/kernel/git/stable/linux-stable.git/tree/drivers/char/random.c?id=refs/tags/v3.15.6#n52 (visitado 04-05-2016).

[36] James Michaelson y col. «Estimates of the Sizes at Which Breast Cancers Become Detectable on Mammographic and Clinical Grounds:» en. En: *Journal of Women's Imaging* 5.1 (feb. de 2003), págs. 3-10. ISSN: 1084-824X. DOI: 10.1097/00130747-200302000-00002.

[37] Frank L. Meyskens, Stephen P. Thomson y Thomas E. Moon. «Quantitation of the number of cells within tumor colonies in semisolid medium and their growth as oblate spheroids». En: *Cancer research* 44.1 (1984), págs. 271-277. URL: http://cancerres.aacrjournals.org/content/44/1/271.short.